
Project submitted in partial fulfilment of the requirements for the M.Sc. degree in Tel-Aviv University, in Electrical Engineering

by

Eliezer Yucht

Prepared under the supervision of Prof. Dana Ron.

December 2016
Abstract

We present an estimation of the closeness level between the distribution of RC4 keystream bytes (at some predefined locations in the stream) to a uniform distribution (over the same domain). Our estimation is based on the $L_1$ distance between a pair of distributions over the same domain.

Although learning the distance (empirically) between a distribution of a pair of consecutive bytes in the stream to the uniform distribution over the same domain ($N = 2^{16}$), to $\epsilon$ accuracy, requires (theoretically) a sample size of $S = O(N^2)$, there are tests that determine (with high probability) whether a given distribution over an $N$-element domain is a uniform distribution or $\epsilon$ farther (in the $L_1$ sense) from the uniform distribution. Such tests require a sample size of $S = O(\sqrt{N^3})$.

We have managed to show (empirically) that one may distinguish between distributions with different levels of distance (from the uniform distribution), even with a sample size of $2^{22}$ samples, via a collision tester. This test counts the number of colliding pairs in the sample. Therefore, we reduce the execution time from (approximately) 10 days for estimating the $L_1$ distance via learning to less than 25 minutes under the collision tester (on a single CPU).

In addition, we apply another uniformity test (the Paninski test) to our distributions (four pairs of consecutive bytes in a RC4 keystream at different locations) and analyze the results. Finally, we present the fingerprints of these distributions when $2^{21}$ samples are used.
B. Source Code

B.1 Learning code ................................. 48
   B.1.1 RC4_learn.cpp ............................ 48
   B.1.2 TKIP_learn.cpp ......................... 51
   B.1.3 Learn_RC4_TKIP.m (Matlab script) .... 54

B.2 Paninski test code ............................... 56
   B.2.1 RC4_Paninski_test.cpp .................. 56
   B.2.2 TKIP_Paninski_test.cpp ................. 59
   B.2.3 Paninski_test.m (Matlab script) ...... 62

B.3 Collision tester code .......................... 63
   B.3.1 RC4_collision_test.cpp ................. 63
   B.3.2 TKIP_collision_test.cpp ............... 66
   B.3.3 Random_collision_test.cpp ............. 69
   B.3.4 Collision_histograms.m (Matlab script) 71

B.4 Fingerprint code ................................ 77
   B.4.1 RC4_fingerprint.cpp ..................... 77
   B.4.2 TKIP_fingerprint.cpp ................... 80
   B.4.3 Fingerprint_tester.m (Matlab script) 83
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Pseudo code for implementing RC4</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Empirical distribution of $Z_1$ (using $2^{24}$ random keys)</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Empirical distribution of $Z_2$</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Empirical distribution of $Z_{100}$</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Empirical distribution of $Z_{101}$</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Empirical distribution of $TK_1$ when $(TSC_0, TSC_1)$ is equal to (a) $(0x00, 0x00)$ and (b) $(0x00, 0x20)$</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Empirical distribution of (a) $TK_{17}$ and (b) $TK_{33}$ when $(TSC_0, TSC_1)$ is equal to $(0x00, 0x00)$</td>
<td>11</td>
</tr>
<tr>
<td>6.1</td>
<td>The number of colliding pairs for the four distributions, using a sample size of $S = 2^{18}$</td>
<td>26</td>
</tr>
<tr>
<td>6.2</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{18}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$</td>
<td>27</td>
</tr>
<tr>
<td>6.3</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{20}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$</td>
<td>28</td>
</tr>
<tr>
<td>6.4</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{20}$, of the distributions $(Z_1, Z_2)$ and $(TK_1, TK_2)$</td>
<td>29</td>
</tr>
<tr>
<td>6.5</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{22}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$</td>
<td>30</td>
</tr>
<tr>
<td>6.6</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{24}$, of the distributions $(Z_4, Z_5)$, $(Z_{100}, Z_{101})$ and $U$</td>
<td>31</td>
</tr>
<tr>
<td>6.7</td>
<td>The number of colliding pairs, using a sample size of $S = 2^{26}$, of the distributions $(Z_4, Z_5)$, $(Z_{100}, Z_{101})$ and $U$</td>
<td>32</td>
</tr>
<tr>
<td>7.1</td>
<td>The average fingerprints (entries 15-55) of our four distributions</td>
<td>36</td>
</tr>
<tr>
<td>7.2</td>
<td>The average fingerprints (entries 14-21) of our four distributions</td>
<td>36</td>
</tr>
<tr>
<td>7.3</td>
<td>The average fingerprints (entries 22-27) of our four distributions</td>
<td>37</td>
</tr>
<tr>
<td>7.4</td>
<td>The average fingerprints (entries 28-36) of our four distributions</td>
<td>37</td>
</tr>
<tr>
<td>7.5</td>
<td>The average fingerprints (entries 37-46) of our four distributions</td>
<td>38</td>
</tr>
<tr>
<td>7.6</td>
<td>The average fingerprints (entries 47-56) of our four distributions</td>
<td>38</td>
</tr>
<tr>
<td>7.7</td>
<td>The average fingerprints (entries 57-84) of our four distributions</td>
<td>39</td>
</tr>
<tr>
<td>7.8</td>
<td>The average fingerprints (entries 210-270) of our four distributions</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Measured $L_1$ distance, using a sample size of $S = 2^{37}$ ........................................ 19
4.2 Measured $L_1$ distance, using a sample size of $S = 2^{38}$ ........................................ 20

5.1 Paninski test results, using a sample size of 60,000 ................................................. 23

6.1 Collision tester results, using a sample size of $2^{18}$ ................................................ 27
6.2 Collision tester results, using a sample size of $2^{20}$ ................................................ 29
6.3 Collision tester results, using a sample size of $2^{22}$ ................................................ 30
6.4 Collision tester results, using a sample size of $2^{28}$ ................................................ 32
6.5 Collision tester results, using a sample size of $2^{32}$ ................................................ 33
Chapter 1

Introduction

RC4 is a stream cipher. A few years ago, it was highly popular and integrated in many communication systems and protocols such as WPA-TKIP and SSL. However, over the years, multiple vulnerabilities have been discovered in it. In particular, strong statistical biases (from the uniform distribution) were found in RC4 keystream bytes (especially in the initial keystream bytes) [AlF+13; FM00; Iso+13; MS01; PPS14; VP15] (see also Chapter 2 for further details).

In this project, we estimate the closeness between the distribution of RC4 keystream bytes (at some predefined locations in the stream) and the uniform distribution (over the same domain). We use the $L_1$ distance as a measure for this purpose. Estimating the $L_1$ distance (empirically) between some distribution and the uniform distribution, over a domain with $N$ elements, to $\epsilon$ accuracy requires (theoretically) a sample size of $S = O(\frac{N}{\epsilon^2})$ (see also Section 2.4 and a proof in Appendix A).

A problem arises when $N$ is too large (for example, in Big Data) and/or $\epsilon$ is too small (when better accuracy is required). In those cases, one cannot endure such sample size complexity (which directly influences the time of execution).

We suggest to use the "collision tester" as a tool for estimating the distance between some distribution and the uniform distribution (over the same domain). With this tester, one can improve the required sample size complexity, thus (significantly) reducing the time of execution.

1.1 Project goals

We want to estimate (qualitatively) the level of closeness between the distribution of RC4 keystream bytes (at some predefined locations in the stream and also when TKIP is used) and the uniform distribution.

Specifically, we would like to be able to determine which bytes in the keystream are "good" for encryption (i.e., relatively close to the uniform distribution) and which bytes are not recommended for encryption use (i.e., farther than some threshold from the uniform distribution).

We use some tools from property testing; thus, on the one hand, we can settle for a smaller sample size (relative to $S = O(\frac{N}{\epsilon^2})$) while, on the other hand, still being able to distinguish well between different levels of closeness to the uniform
The following are the four distributions that we use (see Chapter 3 for a further explanation of these notations):

1. \((Z_1, Z_2)\) - The first two bytes in the RC4 keystream (these bytes have a relatively strong bias; see Section 2.2).

2. \((Z_{100}, Z_{101})\) - The pair of bytes at locations 100 and 101 in the RC4 keystream. (We believe that, at these locations, the bias is weaker, as previous results have also shown this; see Section 2.2).

3. \((TK_1, TK_2)\) - The first two bytes in the (RC4) TKIP keystream, where \((TSC_0, TSC_1) = (0x00, 0xFF)\) (see Section 2.3 for further details).

4. \((TK_{32}, TK_{33})\) - The pair of bytes at locations 32 and 33 in the (RC4) TKIP keystream, where \((TSC_0, TSC_1) = (0x00, 0x00)\) (see Section 2.3).

Each of the (above) distributions is actually a joint distribution of a pair of consecutive bytes in the keystream. Thus, our domain can be defined as the set \(\{0, 1, \ldots, 2^{16} - 1\}\) (for further details and examples, see Chapter 3). In particular, its size is \(N = 2^{16}\).

This project can be divided into two main parts: In the first part (Chapter 4), we obtain an estimation (empirically and with high probability), with a sample size of \(S = O\left(\frac{N}{\epsilon^2}\right)\), to the distance (in the \(L_1\) sense) between each of the above distributions to the uniform distribution. These results will be used as a reference to the closeness level of the four distributions relative to the uniform distribution.

In the second part (Chapters 5-7), we apply uniformity tests (see also Section 2.5) to these four distributions. These tests are based on uniformity testing, which obtains access to samples from an unknown distribution (but over a known domain), and checks whether this (tested) distribution is the uniform distribution or if it is farther (than some threshold) from the uniform distribution.

After we obtain the results, we check whether one can distinguish between the distributions (according to their closeness level from the uniform distribution, which we had achieved from the first part) via the sample that we have used. If so, we continue and apply the test on a smaller sample, etc. Then, we will be able to reduce the complexity of the sample size and therefore significantly improve the execution time.
1.2 Project organization

Here, we provide a brief description of the following chapters in this project:

In Chapter 2, we give a background on the RC4 cipher, its biases, and in particular its usage in the WPA-TKIP protocol. We introduce the $L_1$ distance (measure) and the fingerprint concept. In addition, we describe some uniformity tests.

In Chapter 3, we introduce definitions and notations that we have used throughout this report.

In Chapter 4, we give the results obtained while learning the $L_1$ distance for each measured distribution (from the above four) to the uniform distribution ($\|D_{(z_r, z_{r+1})}, U_{2^{16}}\|_1$) using a sample size of $S = O(\frac{N}{\epsilon^2})$.

In Chapter 5, we present a uniformity test (Paninski test), which, given an unknown distribution, determines (with high probability) either that the distribution is the uniform distribution or that it is $\epsilon$ farther than the uniform distribution over the known domain, using a sample size of $S = O(\frac{N}{\epsilon^2})$. We apply this test to each of the four distributions and discuss the results obtained.

In Chapter 6, we present a test that counts the number of colliding pairs in a sample. This test is given an unknown distribution and determines (with high probability) either that the distribution is the uniform distribution or that it is $\epsilon$ farther than the uniform distribution (over the known domain) using a sample size of $S = O(\frac{N}{\epsilon^4})$. We apply this test to each of the four distributions, with a few samples (each time with different sample sizes), and discuss the results.

In Chapter 7, we give the fingerprints of the four distributions, which are obtained using a sample size of $S = 2^{21}$, and discuss the results.

In Chapter 8, we conclude and suggest a few directions for future work.

\footnote{Recently, Diakonikolas et al. \cite{Dia+16} managed to show that even $S = O(\frac{N}{\epsilon^2})$ is sufficient (similar to the size required by the Paninski test, this size is also the lower bound for this task).}
Chapter 2

Background

We give a brief background on the following topics: the RC4 cipher and its usage in WPA-TKIP, RC4 keystream biases, the $L_1$ distance between two distributions, uniformity testing and the fingerprint concept.

2.1 RC4 cipher

RC4 is a stream cipher developed by Ron Rivest for "RSA Security". In 1994, the code was leaked and eventually found its way to the Internet and then to the wider public. The cipher implementation is simple, and it is known to be very fast in software (and hardware). These features gained RC4 its popularity and its integration in many systems and communication protocols such as WEP, WPA-TKIP (wireless communication protocols) and SSL and TLS (secure communication protocols over TCP/IP).

However, over the years, many vulnerabilities have been found in RC4, especially when its first bytes in the keystream are used (and not discarded); in 2015, IETF requested prohibiting the use of RC4 in TLS. [Pop15]

The cipher is composed of the internal state (which we denote by $S$) and two indexes. The state consists of a permutation of the set $\{0, 1, \ldots, 255\}$ (which contains all the possible values a single byte can have). The two indices $i, j$ satisfy $0 \leq i, j \leq 255$.

Similarly, one can describe $S$ as a 256-byte array, where each cell in the array contains a unique byte value, and the two indices $(i, j)$ point to a specific cell/location in the array.

The RC4 algorithm consists of two steps as follows:

1. Initialization step - Key Scheduling Algorithm (KSA).

In the initialization step, the state $S$ is initialized with a symmetric key (the key does not have to be a certain length; however, it is common to use a key whose length is in the range of 5 to 32 bytes, namely, between 40 to 256 bits).
In the second step, the keystream bytes are generated [according to the message (plaintext) length that we want to encrypt (we denote by $m$ the message length in bytes)].

During each round (denoted by $r = 1, 2, \ldots, m$), a call to the PRGA routine is made, which updates the state $S$ and generates a new byte for the keystream (denoted by $Z_r$). This byte is XORed (bitwise) with the plaintext byte (at the same location), and the output is the ciphertext byte. More formally, given the message bytes (denoted by $P_r$, where $r = 1, 2, \ldots, m$) and the keystream bytes, $Z_r$ (generated in the PRGA step), the output ciphertext bytes, $C_r$, are calculated by $C_r = P_r \oplus Z_r$.

In Figure 2.1, an implementation of RC4 (the above two steps) in Python-like pseudo code is given. All additions in the code are performed modulo 256.

---

Figure 2.1: Pseudo code for implementing RC4

```python
Listing (1) RC4 Key Scheduling (KSA).
1  j, S = 0, range(256)
2  for i in range(256):
3    j += S[i] + key[i % len(key)]
4    swap(S[i], S[j])
5  return S

Listing (2) RC4 Keystream Generation (PRGA).
1  S, i, j = KSA(key), 0, 0
2  while True:
3    i += 1
4    j += S[i]
5    swap(S[i], S[j])
6    yield S[S[i] + S[j]]
```

---

2This photograph is taken from the article: "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS" [VP15].
2.2 Biases in RC4 keystream bytes

For a cipher to have the Perfect secrecy property (and thus be robust against every adversary even with unlimited computational resources), the key that is used to XOR the message’s bytes has to be chosen randomly, its length must be at least as long as the message’s length, and it can be used only one time (such as OTP).

Unfortunately, in practice, it is very difficult (and typically impossible) to meet the above requirements for the following reasons:

1. Computers cannot generate (truly) random numbers.
2. For very long messages (e.g., 1 G file size), we have to transmit and maintain secretly a key of (at least) that size.

Thus, in practice, all modern ciphers do not have the Perfect secrecy property, and they can only strive for this goal (and address a strong but limited adversary).

Therefore, in stream ciphers (such as RC4), we would like every byte in the keystream to distribute uniformly over all possible byte values (i.e., uniform distribution over the domain \([0, 1, \ldots, 255]\)) and to not be dependent on any other byte in the stream (i.e., \(Z_r\) and \(Z_t\) should be independent \(\forall r \neq t\)).

In RC4, several strong statistical biases (from the uniform distribution) and also dependencies between some bytes in the stream have been found (especially at the initial bytes in the keystream). Among other things, Mantin and Shamir [MS01] found that the second keystream byte (i.e., \(Z_2\)) is twice as likely to be zero compared to uniform, namely, \(\Pr[Z_2 = 0] \approx 2 \cdot 2^{-8}\), where the probability is over the random choice of the key.

AlFardan et al. [AlF+13] showed empirically that all 256 first bytes in the RC4 keystream are biased by estimating their probabilities when 16-byte keys\(^3\) are used.

In the following Figures (2.2-2.5), the probability functions (obtained empirically by \(2^{44}\) random 16-byte keys [AlF+13]) at different locations in the stream are depicted.

---

\(^3\)In many usages of RC4 (e.g., WPA-TKIP and TLS-RC4) a 16-byte key is used.

\(^4\)These figures are taken from "http://www.isg.rhul.ac.uk/tls/biases.pdf".
$Z_1$ (the first keystream byte) empirical distribution:

![Keystream distribution at position 1](image1)

Figure 2.2: Empirical distribution of $Z_1$ (using $2^{44}$ random keys)

$Z_2$ (the second keystream byte) empirical distribution (notice the strong bias toward the '0' value):

![Keystream distribution at position 2](image2)

Figure 2.3: Empirical distribution of $Z_2$
With further stream locations, the bias power is weakened (The exceptions are the bytes where their locations are multiplicative of 16. Apparently this phenomenon is related to the key size).

$Z_{100}$ (the 100th keystream byte) empirical distribution:

Figure 2.4: Empirical distribution of $Z_{100}$

$Z_{101}$ (the 101st keystream byte) empirical distribution:

Figure 2.5: Empirical distribution of $Z_{101}$

In addition to the above biases (which have been relative to a single byte), a strong statistical dependency between $Z_1$ and $Z_2$ has been found \cite{Iso+13}, especially toward the value ‘0’: $\Pr[Z_1 = Z_2 = 0] \approx 3 \cdot 2^{-16}$. 
In addition, Fluhrer and McGrew [FM00] found that pairs of consecutive keystream bytes (i.e., \((Z_r, Z_{r+1})\)) are biased toward specific values throughout the whole keystream (i.e., not only at the first bytes).
For example (when the index \(i = 1\)), \(\Pr\left[(Z_r, Z_{r+1}) = (0, 0)\right] \approx 2^{-16} \cdot (1 + 2^{-7})\).

### 2.3 WPA-TKIP

In 2002, the Wi-Fi Alliance developed the WPA (Wi-Fi Protected Access) protocol to secure wireless networks and replace the previous protocol (WEP), which had suffered from major security vulnerabilities.

To apply the changes to wireless routers with old hardware and develop a fast solution that could be implemented through software update alone, it was decided, in the first stage, to preserve the cipher (RC4) used in WEP and improve only the critical aspects of the protocol, later switching to an improved protocol (WPA2) with another cipher.

The improvements (that were implemented in the first stage) include switching to a temporary 16-byte key and replacing it for each transmitted packet (namely, creating and using a new key for every transmitted frame) as well as adding an integrity protocol, hence the name of the protocol: WPA-TKIP (Temporal Key Integrity Protocol).

For a new packet destined to be transmitted into a WPA-TKIP-protected network, its content is encrypted via RC4. The 16-byte key used for that purpose, denoted by \(K = (K_0, K_1, \ldots, K_{15})\), is generated by a Key Mixing Function (KM) that takes as input three parameters as follows:

1. Temporal encryption key (TK) - a 128-bit (symmetric) key (agreed up and frequently replaced between the Access Point and the client).
2. Transmitter MAC address (TA) - the transmitter’s MAC address (48 bits long).
3. TKIP sequence counter (TSC) - a 48-bit counter that is incremented after each transmitted packet (thus, guaranteeing a different encryption key for every frame).

We denote the 6 bytes of the counter by \(TSC = (TSC_0, TSC_1, \ldots, TSC_5)\), where \(TSC_0\) is the least significant byte.

The key mixing function (KM) outputs a 16-byte key (namely, we can write this as \(K \leftarrow KM(TK, TA, TSC)\)).

To avoid weak-key attacks that broke WEP, the first three bytes of \(K\) are set to [IEEE12]

\[\begin{align*}
    &a) \quad K_0 = TSC_1 \\
    &b) \quad K_1 = (TSC_1 | 0x20) \& 0x7F \\
    &c) \quad K_2 = TSC_0
\end{align*}\]

As a result (and the fact that \(TSC\) is transmitted as plain-text, i.e., it is not encrypted), everyone can find the first three bytes of \(K\).
The remaining 13 bytes (of $K$) are determined by the KM and can be modeled as uniformly random.

In addition, strong biases (at some locations in the keystream) that depend on the pair $(\text{TSC}_0, \text{TSC}_1)$ have been found [PPS14]. Those biases (among other things) led to a recent attack (based only on a broadcast attack) that can break WPA-TKIP within an hour [VP15].

In the following Figures (2.6-2.7) [PPS14], one can see (in blue) the probability functions at some locations in a TKIP keystream. Those probabilities are obtained empirically using $2^{36}$ 13-byte random keys per particular $(\text{TSC}_0, \text{TSC}_1)$ pair. In red (for reference), the probability function appears when all 16 bytes of the key are randomly chosen (such as in TLS-RC4). As you can see, when the three bytes of the key are fixed (according to some value of $(\text{TSC}_0, \text{TSC}_1)$) [such as in TKIP, the ”blue case’’], stronger biases are created.

The probability function of $TK_1$ (the first byte in the TKIP keystream) when the pair value of $(\text{TSC}_0, \text{TSC}_1)$ is equal to (a) $(0\times00, 0\times00)$ and (b) $(0\times00, 0\times20)$ is shown in Figure 2.6.

![Figure 2.6: Empirical distribution of $TK_1$ when $(\text{TSC}_0, \text{TSC}_1)$ is equal to (a) $(0\times00, 0\times00)$ and (b) $(0\times00, 0\times20)$](image)
The probability function of (a) $TK_{17}$ and (b) $TK_{33}$ when $(TSC_0, TSC_1) = (0x00, 0x00)$ is shown in Figure 2.7.

Figure 2.7: Empirical distribution of (a) $TK_{17}$ and (b) $TK_{33}$ when $(TSC_0, TSC_1)$ is equal to $(0x00, 0x00)$.

### 2.4 The $L_1$ measure

We introduce the $L_1$ distance as a measure of distance between two (discrete) distributions $p, q$. Let $p, q$ be two (discrete) probability functions over the domain $D$; then, the $L_1$ distance between them, which we denote by $\|p, q\|_1$, is defined as the sum of the distances between their probabilities over all the domain elements, namely,

$$\|p, q\|_1 \triangleq \sum_{x \in D} |p(x) - q(x)|$$

First, notice that $0 \leq \|p, q\|_1 \leq 2$ (the lower bound is due to a sum of non-negative values, and the upper bound is flowing from the definitions of $p, q$ as probability functions).

In addition, in the special case where we are interested in calculating the $L_1$ distance between some distribution $p$ and the uniform distribution over the domain $[N] \triangleq \{0, 1, \ldots, N - 1\}$, the following holds:

$$\|p, U_N\|_1 = \sum_{x \in [N]} \left| p(x) - \frac{1}{N} \right|$$

Sometimes, the distribution $p$ is unknown explicitly (such as in the case of the RC4 keystream bytes’ distributions at different locations in the stream, where the distribution is induced by choosing random 16-byte keys; in such a case, for calculating $p$ explicitly, we need $2^{128}$ samples in each location, which is infeasible).

In those cases, we can only estimate the distance $\|p, U_N\|_1$ using a sample with $S$ elements (which is induced by $p$).

We denote by $\hat{p}$ the empirical distribution from the sample.

Namely, given $S$ samples, $x_1, x_2, \ldots, x_S$ (which are drawn independently according to $p$ over $[N]$),
we define the probability function $\hat{p}$ as the relative frequency of each of the values $i \in [N]$ in the sample; more formally,

$$\forall i \in [N], \quad \hat{p}(i) \triangleq \frac{|\{j \in \{1, 2, \ldots, S\} : x_j = i\}|}{S}$$

Now, since $\hat{p}$ is well defined, we can easily calculate $\|\hat{p}, U_N\|_1$. If $S = O(\frac{N}{\epsilon^2})$, then with high probability, $\hat{p}$ is $\epsilon$ close to $p$ (i.e., $\|p, \hat{p}\|_1 \leq \epsilon$; see the proof in Appendix A). Therefore, using the triangle inequality, it can be concluded that

$${\text{max}}(0, \|\hat{p}, U_N\|_1 - \epsilon) \leq \|p, U_N\|_1 \leq \|\hat{p}, U_N\|_1 + \epsilon$$

(2.1)

In this way, we can obtain a good estimation (to $\epsilon$ accuracy) to $\|p, U_N\|_1$.

As mentioned previously, the ideal distribution for the keystream bytes (in any stream cipher and particularly in RC4) is the uniform distribution (in addition, we would like for any two bytes at different locations in the stream to be independent). Since we would like the keystream bytes to distribute uniformly (or at least very close to uniformly), the smaller the $L_1$ distance that we can obtain (between a keystream byte at a specific location in the stream and the uniform distribution over the same domain), the better the situation (since this means that the distribution of this byte is closer to the uniform distribution).
2.5 Uniformity testing

As mentioned in the previous section, for estimating the $L_1$ distance between some distribution $p$ to the uniform distribution over the same domain ($[N]$), to $\epsilon$ accuracy, a sample size of $S = O(\frac{N}{\epsilon^2})$ is required.

In many cases, the accurate distance is not important, and one only needs to decide that either a given distribution is the uniform distribution or it is $\epsilon$ away (in the $L_1$ sense) from the uniform distribution.

For those cases, there exist tests (called uniformity tests) that, with a sample size of $S = O(\sqrt{\frac{N}{\epsilon^2}})$, can determine (with high probability) which of the above two options is correct. We will now briefly present two such tests.

2.5.1 Counting samples that are not colliding in a sparse sample

In this test, we take a ”sparse sample” (where $S < N$) and count how many samples appeared exactly once in the sample (i.e., did not collide with any other sample). We denote this value by $K_1$. The test is based on the idea that the farther a distribution is from the uniform distribution, the more collisions that would be seen among its samples (namely, a pair of (different) samples $1 \leq i < j \leq S$ with the same value: $x_i = x_j$). Thus, fewer samples would not collide at all.

Therefore, if we obtain $K_1$ that is below some threshold, the test will reject the claim that the tested distribution is the uniform distribution (and otherwise will accept the claim). This test was first presented by Liam Paninski [Pan08] (in this report, we sometimes refer to it as the ”Paninski test”).

The test required a sample size of $S = O(\sqrt{\frac{N}{\epsilon^2}})$; this size is also the lower bound for this task (uniformity testing), as was proved in that article.

2.5.2 Collision tester

Similar to the Paninski test, this test is also based on the idea that, the farther a distribution is from the uniform distribution, the more likely that the collision probability (of that distribution) will be higher. However, here, the test counts the number of pairs that collide in the sample (for estimating the collision probability of the tested distribution).

We denote this value by $C_D$, namely,

$$C_D = |\{(i, j); \ 1 \leq i < j \leq S : x_i = x_j\}|$$

If this value is above some threshold, the test will reject the claim that the tested distribution is the uniform distribution (otherwise, it will accept the claim).

This test required a sample size of $S = O(\sqrt{\frac{N}{\epsilon^2}})$ [GR00], although this size is larger than the lower bound\footnote{See the remark in footnote 1 above.} ($0 < \epsilon < 1$). This test has an advantage that it is not limited to the sparse case (such as the case in the Paninski test) and can also be applied to the general case (when $S > N$).
2.6 Symmetric properties and fingerprints

Symmetric properties of distributions are properties that are invariant under relabeling of the domain elements. More formally, let \( p \) be a distribution over the domain \([N] = \{0, 1, \ldots, N-1\}\). For each permutation \((\pi : [N] \rightarrow [N])\), define the following distribution, \( p_\pi : \forall i \in [N], \quad p_\pi(i) \triangleq p(\pi(i)) \). Then, a property \( PROP \) (of distribution \( p \)) will be called a symmetric property if the following holds for all \( p_\pi \) distributions:

\[
PROP(p) = PROP(p_\pi)
\]

For example, entropy (of the distribution) and the \( L_1 \) distance between some distribution and the uniform distribution are symmetric properties.

When testing a symmetric property on a distribution using a sample, the only useful information from the sample is its collision statistics. The collision statistics can be derived from the sample’s fingerprint. A fingerprint is a vector whose \( i \)th entry denotes the number of (domain) elements that appear exactly \( i \) times in the sample (i.e., experience \( i \)-way collisions) [Val11].

More formally, given a sample with \( S \) samples, \( x_1, x_2, \ldots, x_S \), for each \( i \in [N] \), we denote by \( \#_i(x_1, x_2, \ldots, x_S) \) the number of samples that fell into the \( i \)th bin, namely,

\[
\#_i(x_1, x_2, \ldots, x_S) = |\{k \in \{1, 2, \ldots, S\} : x_k = i\}|
\]

We denote by \( F = (F_1, F_2, \ldots, F_S) \) the obtained fingerprint, where \( F_j \) counts the number of elements that appear exactly \( j \) times in the sample, namely,

\[
F_j = |\{i \in [N] : \#_i(x_1, x_2, \ldots, x_S) = j\}|
\]

For example, suppose that we rolled a dice ten times and obtain the following results: \((1, 2, 1, 1, 5, 5, 2, 6, 1, 3)\). We can depict the obtained results using a histogram over the domain \( \{1, 2, \ldots, 6\} \) as follows: \((4, 2, 1, 0, 2, 1)\). Here, the value ‘1’ appeared four times in the experiment, the value ‘2’ appeared twice, ‘3’ appeared only once, ‘4’ did not appear at all, ‘5’ appeared twice, and ‘6’ appeared once. Therefore, the obtained fingerprint is \((2, 2, 0, 1)\), which indicate that two elements appeared (in the sample) once (‘3’ and ‘6’), two elements appeared twice (‘2’ and ‘5’), no element appeared only three times, and one element appeared four times (‘1’) (the fingerprint is the histogram of the histogram).

Notice that, for measuring the \( L_1 \) distance between the tested distribution and the uniform distribution (a symmetric property, as mentioned above), the only relevant information (from the experiment) is the fingerprint. After all, even if we were to replace ‘4’ with ‘5’ in our experiment’s results (i.e., \(1, 2, 1, 1, 4, 4, 2, 6, 1, 3\)), we would obtain the same fingerprint \((2, 2, 0, 1)\) and the same \( L_1 \) distance (from the uniform distribution over the domain \( \{1, 2, \ldots, 6\} \)).
Chapter 3
Notations and Definitions

In this report, we use the following notations and definitions, unless otherwise mentioned explicitly.

We denote by \([N]\) the set of all integers between 0 and \(N - 1\) (inclusive):
\[ [N] \triangleq \{0, 1, \ldots, N - 1\}. \]
We define the (discrete) probability function \(p\) over the domain \([N]\) as the function
\[ p : [N] \to [0, 1] \] such that
\[ \sum_{i=0}^{N-1} p(i) = 1. \]
We use \(U_N\) to denote the uniform distribution over \([N]\) (in this distribution, \(p(i) = \frac{1}{N}, \ \forall i \in [N]\)).
Sometimes, we refer to the domain elements \((i \in [N])\) as “bins”.

Let \(p, q\) be two (discrete) probability functions over the domain \(D\); then, the \(L_1\) distance between them (denoted by ||\(p, q||_1\)) is defined as the sum of distances between their probabilities over all domain elements, namely,
\[ ||p, q||_1 \triangleq \sum_{x \in D} |p(x) - q(x)| \]
Unless otherwise stated, when we use terms of distance (e.g., closer and farther) regarding distributions, the terms refer to the \(L_1\) distance.

For a probability function \(p\), we denote by \(\hat{p}\) the empirical distribution learned using \(S\) samples, \(x_1, x_2, \ldots, x_S\) (drawn randomly and independently) according to \(p\), namely,
\[ \forall i \in D, \ \hat{p}(i) \triangleq \frac{|\{j \in \{1, 2, \ldots, S\} : x_j = i\}|}{S} \]
We indicate by \(F_D = (F_1, F_2, \ldots, F_S)\) the fingerprint (see Section 2.6 for details) derived from a sample size of \(S\) according to the distribution \(D\).
When we refer to the \(i\)th element in some fingerprint, we use either \(F_i\) or \(F(i)\).

Throughout this project, we use an RC4 cipher with 16-byte initial key \(K\).
The key bytes are denoted by \(K_0, K_1, \ldots, K_{15}\), where \(K_0\) is the first byte of the key.

The keystream bytes are denoted by \(Z_r\), where \(r = 1, 2, \ldots\) indicates the byte location in the stream (e.g., the first keystream byte is denoted by \(Z_1\)).
For distinguishing between “regular” use of RC4 and the case in which TKIP is used
(see Section 2.3), we denote by $TK_r$ the keystream bytes (when TKIP is used). As before, $r$ indicates the location of the byte in the stream.

We denote by $D_{Z_r.Z_q}$ (or by $D_{TK_r.TK_q}$) the joint distribution (induced by the random choice of the key $K$) of the keystream byte pair $(Z_r, Z_q)$ (in this project, we are interested in the case where $q = r + 1$, i.e., consecutive keystream bytes). To simplify the notation, we often use $(Z_r, Z_{r+1})$ (or $(TK_r, TK_{r+1})$ when using TKIP) to refer to the joint distribution.

For example,

- $(Z_{100}, Z_{101})$ is the joint distribution of the RC4 keystream bytes at the 100th and 101st locations in the stream.
- $(TK_{32}, TK_{33})$ is the joint distribution of the TKIP keystream bytes at the 32nd and 33rd locations in the stream.

Notice that, since we use pairs of bytes, we can map the domain of the joint distribution $(Z_r, Z_{r+1})$ to the domain $\{0, 1, \ldots, 2^{16} - 1\}$. We do this by referring to the pair of bytes as a decimal number stored in the computer’s memory (occupying two bytes of space); without loss of generality, we choose $Z_r$ to be the least significant byte.

The following are few examples of the above map for various values of $(Z_r, Z_{r+1})$:

- $(0x00, 0x00) \rightarrow 0$
- $(0xFF, 0x00) \rightarrow 255$
- $(0x00, 0xFF) \rightarrow 65, 280$
- $(0xFF, 0xFF) \rightarrow 65, 535 = 2^{16} - 1$

When the program’s execution time is mentioned, we will use the following format: $[xx]h.[yy]m.[zz]s$.

Here, $[xx]$ indicates how many hours (when this value is greater than zero) the program ran, $[yy]$ indicates how many minutes the program ran, and $[zz]$ indicates how many seconds the program ran.

For example,

- To indicate (a time of execution of) 2 minutes and 43 seconds, we say $2m.43s$.
- To indicate (a time of execution of) 3 hours, 35 minutes and 9 seconds, we say $3h.35m.9s$.

In addition, we note that all our C++ programs (see the source code in Appendix B) was compiled with $g++$ version 4.9.2, and all the simulations ran on a desktop computer with an Intel(R) Core(TM) i5-4430 processor (with 2 cores) on a Linux 64-bit operation system.
Chapter 4

Estimating the $L_1$ distance via learning

We would like to obtain an estimation of the $L_1$ distance between the induced distribution (by random choice of 16-byte keys) of RC4 keystream bytes (at different locations) and the uniform distribution.

As mentioned in Section 2.2, the ideal situation is when the keystream bytes of a (stream) cipher are uniformly distributed and without dependencies among them (to achieve perfect secrecy). Thus, a stream cipher will be considered as better (than another stream cipher) when the $L_1$ distance between its keystream byte distribution and the uniform distribution is smaller.

To achieve an accurate estimation of the $L_1$ distance, we use a learning algorithm. Our algorithm draws (randomly and independently) $S$ samples from the tested distribution. For each domain element $i \in [N]$, we count how many times it appears in our sample (and denote this value by $y_i$). Then, we can estimate the probability of "falling" into the $i$th bin (denoted by $\hat{p}(i)$) by calculating $\hat{p}(i) = \frac{y_i}{S}$.

Finally, we calculate the distance between the tested distribution (denoted by $\hat{D}$) and the uniform distribution (over the same domain, denoted by $U_N$) using the following formula:

$$\|U_N, \hat{D}\|_1 = \sum_{i \in [N]} |\hat{p}(i) - \frac{1}{N}|.$$

In this project, we measure the joint distribution of a pair of consecutive bytes in the keystream (i.e., $(Z_r, Z_{r+1})$); thus, the distribution’s domain is equal to $[N] = [2^{16}]$ (see Chapter 3 for further explanation). Therefore, we calculate:

$$\|U_{2^{16}}, \hat{D}\|_1 = \sum_{i \in [2^{16}]} |\hat{p}(i) - 2^{-16}|.$$

As mentioned in Section 2.4, to estimate (with high probability) the $L_1$ distance to $\epsilon$ accuracy, a sample size of (at least) $S = O\left(\frac{N}{\epsilon^2}\right)$ is required.

It can be shown (see Theorem 1 in Appendix A) that, when using a sample size of $S = \frac{8N}{\epsilon^2}$ (at least with $1 - \frac{1}{\sqrt{8}} \approx 0.65$ probability$^6$), $\hat{p}$ is $\epsilon$ close to $p$ (i.e., $\|p, \hat{p}\|_1 \leq \epsilon$).

Therefore, (using the triangle inequality) the following holds:

$$\max(0, \|\hat{p}, U_N\|_1 - \epsilon) \leq \|p, U_N\|_1 \leq \|\hat{p}, U_N\|_1 + \epsilon$$

$^6$One can reduce the error probability ($\delta$) as desired by repeating this experiment $O(\log \frac{1}{\delta})$ times and choosing the median result (see Theorem 2 in Appendix A).
From our initial tests, we found that the $L_1$ distance obtained from the experiments ($\|\hat{p}, U_N\|_1$) is relatively small (on the order of thousandths); thus, we have to use a (relatively) small $\epsilon$ (to achieve the proper resolution, which enables distinguishing between the tested distributions). However, choosing too small of an $\epsilon$ will result in a larger sample size (and thus a longer execution time). Therefore, we decided (at first) to choose $\epsilon = \frac{1}{2^9} = 2^{-9}$ (and thus obtain a resolution of $1.95 \cdot 10^{-3}$). As a result, we used (at first) a sample size of $S = \frac{8\cdot N}{\epsilon^2} = 2^{37}$.

4.1 Randomly choosing the keys

To use the learning algorithm, we need to generate (randomly and independently) $2^{37}$ samples of the keystream bytes (at the desired location). For each sample, we had to generate a new (random) initial key (for the KSA stage; see Section 2.1). In all our experiments, we used a 16-byte key size (as in WPA-TKIP and TLS-RC4).

To generate the random keys, we use the `<random>` library (under STL since C++11). We use the Mersenne Twister engine to generate the uniform distribution over the desired domain. To create the RC4 (initial) keys, we used a uniform distribution generator over the domain $[2^{64}] = \{0,1,\ldots,2^{64} - 1\}$; thus, the key assignment was performed in two steps as follow: The first half of the key (the first 8 bytes) was initialized by calling the generator once, and the second half of the key (the remaining 8 bytes) was initialized by a second call to the generator (see the code in the function "create_key" in Appendix B.1.1). Specifically,

$$(K_0, K_1, \ldots, K_7) \leftarrow R [2^{64}]$$
$$(K_8, K_9, \ldots, K_{15}) \leftarrow R [2^{64}]$$

To create the TKIP (initial) keys, we performed the following steps: The first three bytes of the key were determined according to the values of $TSC_0$ and $TSC_1$ (recall Section 2.3) and remained constant through the complete program execution. We then used a uniform distribution generator over the domain $[2^{32}] = \{0,1,\ldots,2^{32} - 1\}$, and the remaining (13 bytes) key assignments were performed as follow: The fourth byte (of the key, i.e., $K_3$) was initialized by calling the generator and applying modulo 256 to the output (in effect, this was equivalent to taking the least significant byte from the generator’s output). The remaining 12 bytes of the key were divided into three quartets, where each quartet was initialized by one call to the generator (each call initialized four bytes; see the code in the function "create_key" in Appendix B.1.2). Specifically,

$$(K_3) \leftarrow R [2^8]$$
$$(K_4, K_5, K_6, K_7) \leftarrow R [2^{32}]$$
$$(K_8, K_9, K_{10}, K_{11}) \leftarrow R [2^{32}]$$
$$(K_{12}, K_{13}, K_{14}, K_{15}) \leftarrow R [2^{32}]$$
We remark here that using the (naive) rand() function (from C) does not fit our needs for the following reasons:

- rand() generates random values between 0 and RAND_MAX (in Microsoft Visual Studio, RAND_MAX’s value is only 32,767); therefore, even for generating a single 16-byte key, rand() must be invoked many times.
- The cycle of rand() is only $2^{32}$ (versus $2^{19937}$ in Mersenne Twister). This means that, in many simulations (which require many random samples, such as in our case), we cannot use rand() (since, after every $2^{32}$ calls, we will obtain the same sequence).

### 4.2 Simulation results

We applied the learning algorithm to each of the four measured distributions. We first used a sample size $S = 2^{37}$, which required that our program run for approximately five days on a single CPU.

After generating $2^{37}$ samples, we counted, for each bin $\forall i \in [2^{16}]$, how many times it appeared in the sample ($y_i$) and output the results to a file. Then, we used Matlab (see the code in Appendix B.1.3) to read the data from the file and calculate the obtained $L_1$ distance: $\|U_{2^{16}}, \hat{D}\|_1$.

The results are summarized in the following table:

<table>
<thead>
<tr>
<th>Distribution learned</th>
<th>$|U_{2^{16}}, D|_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>0.00784</td>
</tr>
<tr>
<td>$(Z_{100}, Z_{101})$</td>
<td>0.00073</td>
</tr>
<tr>
<td>$(TK_1, TK_2)$</td>
<td>0.06117</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>0.00481</td>
</tr>
</tbody>
</table>

Table 4.1: Measured $L_1$ distance, using a sample size of $S = 2^{37}$

For $\epsilon = 2^{-9}$ (using Theorem 1 in Appendix A), we can say that, with high probability, the following inequalities (see inequality 2.1 in Section 2.4) hold:

\[
\begin{align*}
0 & \leq \|D_{Z_{100}, Z_{101}}, U_{2^{16}}\|_1 \leq 0.00268 \\
0.00286 & \leq \|D_{TK_{32}, TK_{33}}, U_{2^{16}}\|_1 \leq 0.00676 \\
0.00589 & \leq \|D_{Z_1, Z_2}, U_{2^{16}}\|_1 \leq 0.00979 \\
0.05922 & \leq \|D_{TK_1, TK_2}, U_{2^{16}}\|_1 \leq 0.06312
\end{align*}
\]

As one can see from the results, among the four distributions that we tested, the farthest distribution (in the $L_1$ sense) from the uniform distribution is $(TK_1, TK_2)$, and the closest distribution is $(Z_{100}, Z_{101})$. Unfortunately, the resolution obtained using $\epsilon = 2^{-9}$ is not sufficiently tight to allow us to determine (for sure) which of the distributions, $(Z_1, Z_2)$ or $(TK_{32}, TK_{33})$, is farther from the uniform distribution (as the upper bound of $\|D_{TK_{32}, TK_{33}}, U_{2^{16}}\|_1$ is greater than the lower bound of $\|D_{Z_1, Z_2}, U_{2^{16}}\|_1$).

Therefore, for the distributions $(Z_1, Z_2)$ and $(TK_{32}, TK_{33})$, we decided to increase the size of the sample to $2^{38}$, thus achieving a higher resolution (now, $\epsilon = \frac{2^{-9}}{\sqrt{2}}$) in
the hope that, this time, we will be able to determine which distribution is farther (from the uniform distribution).

As a result, for those distributions, it took our program approximately 10 days to run (on a single CPU).

The following are the results of learning the $L_1$ distance (between each of the above two distributions to the uniform distribution) using a sample size of $S = 2^{38}$:

<table>
<thead>
<tr>
<th>Distribution learned</th>
<th>$|U_{2^{16}}, D|_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>0.00784</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>0.00479</td>
</tr>
</tbody>
</table>

Table 4.2: Measured $L_1$ distance, using a sample size of $S = 2^{38}$

With $\epsilon = 2^{-9}/\sqrt{2} = 2^{-9.5}$, we obtained a higher accuracy for the $L_1$ distance:

$$0.00341 \leq \|D_{TK_{32}, TK_{33}}, U_{2^{16}}\|_1 \leq 0.00617$$
$$0.00646 \leq \|D_{Z_1, Z_2}, U_{2^{16}}\|_1 \leq 0.00922$$

(The remaining inequalities remain unchanged).

Now, we can sort (with high probability) the four tested distributions according to their distances from the uniform distribution (from the farthest to the closest, from left to right):

$(TK_1, TK_2), (Z_1, Z_2), (TK_{32}, TK_{33}), (Z_{100}, Z_{101})$.

### 4.3 Addressing execution time

The main drawback of the learning approach is its excessive execution time (derived from the sample size). As mentioned above, for a sample size of $2^{38}$, it takes our program 10 days to run (on a single CPU).

This issue can be resolved using a distributed network as follows:

First, run the algorithm on multiple CPUs (e.g., $2^7 = 128$ processors), where each processor uses only a fraction of the samples (e.g., when deploying the samples evenly among the processors, each processor uses $2^{38}/2^7 = 2^{31}$ samples).

Then, gather the results (from the 128 output files) and combine them into one database that contains all the samples ($2^{38}$); from the database, we can calculate the $L_1$ distance (as before).

This way, we can achieve a (approximately 128 times) better execution time (in this example, less than two hours, versus 10 days).

However, this solution requires a relatively large amount of resources (processors), which are not always available, and although it improves the execution time, eventually, we still have to use the same sample size (overall, $2^{38}$).

In this project, we attempt to attack the problem from a different direction. In many (practical) cases, it is not essential to know (exactly) the $L_1$ distance between some distribution and the uniform distribution; it is sufficient to determine (with high probability) between one of two options:

Either the tested distribution is relatively "close" to the uniform distribution (namely, its $L_1$ distance is less than some predefined threshold $\epsilon_1$), or it is too "far" (namely,
its $L_1$ distance is greater than another predefined threshold, $\epsilon_2$, such that $0 < \epsilon_1 < \epsilon_2$). [If its distance is neither too close nor too far, namely, between $\epsilon_1$ and $\epsilon_2$, the determination is immaterial, that is, both results are valid and legitimate in such a case].

For example, in our tested distributions, if we set $\epsilon_1 = 0.003$ and $\epsilon_2 = 0.0063$, we would like to accept distributions that are close to the uniform distribution up to a distance $\epsilon_1 = 0.003$, such as $(Z_{100}, Z_{101})$, and to reject distributions that are farther than $\epsilon_2 = 0.0063$ such as $(Z_1, Z_2)$ and $(TK_1, TK_2)$. In addition, if the distance is between $\epsilon_1$ and $\epsilon_2$ (such as $(TK_{32}, TK_{33})$), the algorithm may either accept or reject the distribution.

We are looking for a property testing algorithm that accepts (outputs ‘1’) with high probability if the $L_1$ distance of the tested distribution (from the uniform distribution) is less than $\epsilon_1$ and rejects (outputs ‘0’) (with high probability) if the $L_1$ distance is greater than $\epsilon_2$. Such a test is called a "tolerant test". In addition, we would like the sample size of the testing algorithm to be smaller (significantly) than the sample size of the learning algorithm (smaller than $O\left(\frac{N}{\epsilon_2}\right)$). That way, we can settle for a smaller sample, thus achieving a better execution time.

Unfortunately, in the general case (of the tolerant test), it has been proven that, for a constant $\epsilon$, there is a lower bound of $\Omega\left(\frac{N}{\log N}\right)$ on the required sample size $[VV10]$. However, it is known that a sample size of $O\left(\frac{N}{\log N \cdot (\epsilon_2 - \epsilon_1)^2}\right)$ is sufficient $[VV11]$. Therefore, theoretically, we can improve the execution time by a factor of $\log N = 16$ (relative to the learning algorithm).

We would like to check (empirically), using uniformity tests, whether we can distinguish the different levels of the $L_1$ distances (between the tested distributions and the uniform distribution) using smaller samples. The following are the methods that we have used:

- Counting samples that are not colliding in a sparse sample (Paninski test).
- Counting pairs of colliding samples.
- Comparing the distributions’ fingerprints.

For each method that we applied (to each distribution among the four tested distributions), we would like to check if the obtained results enable us to obtain an estimation (at least qualitative information) of how close (to the uniform distribution) the tested distribution is. In addition, we would like to test the sample size, which is essential for distinguishing (clearly) between the four distributions. In the following chapters, we introduce these methods in detail and analyze the obtained results.
Chapter 5

Counting samples that are not colliding, in a sparse sample (Paninski test)

In 2008, Liam Paninski [Pan08] introduced a test that can distinguish (with high probability) between the uniform distribution (over given domain) and a distribution that is $\epsilon$ (or more) farther away (in the $L_1$ sense). In the same article, he also presented a lower bound and an upper bound for the required sample size. He showed that, on the one hand, there is no test that can distinguish between the above two cases with fewer than $\Omega\left(\frac{\sqrt{N}}{\epsilon^2}\right)$ samples. On the other hand, if $S = O\left(\frac{\sqrt{N}}{\epsilon^2}\right)$, there is such a test.

His test is based on the observation that the further a distribution is from the uniform distribution, the (more likely) greater the number of collisions that will occur in its sample (a pair of samples $i, j$; $(i < j)$ that yield the same domain element $x_i = x_j$ is counted as one collision). Thus, in a "sparse" sample (where the total number of samples is smaller than the domain size, i.e., $S < N$), the further a distribution is from the uniform distribution, the smaller the expected number of bins that have (exactly) one sample in them (denoted by $K_1$). On the other hand, for the uniform distribution, the expectation of $K_1$ is the greatest.

Now, all that remains is to draw from the tested distribution (randomly and independently) $S = O\left(\frac{\sqrt{N}}{\epsilon^2}\right) < N$ samples and count how many bins (domain elements) have exactly one sample that falls in them. If this value ($K_1$) is smaller than some threshold, we reject the hypothesis that the tested distribution is the uniform distribution (otherwise, we accept the hypothesis).

Unfortunately, in our case, we cannot derive much information from this test for the following reasons:

1. The Paninski test can distinguish whether the tested distribution is the uniform distribution or is $\epsilon$ (or more) further away (in the $L_1$ sense). However, we are looking for an algorithm that has the ability to distinguish whether the

---


7In his article [Pan08], he used $m$ to denote the domain size and $N$ to denote the sample size required. Recall that, in our notations, we use $N$ to denote the domain size ($2^{16}$) and $S$ for the sample size.
tested distribution is "close" to the uniform distribution (up to $\epsilon_1$) or farther than $\epsilon_2$ (recall the "tolerant test" in Section 4.3).

2. In our case, all the tested distributions (excluding $(TK_1, TK_2)$) are closer to the uniform distribution by more than 0.01 (see the simulation results in Section 4.2). Therefore, we should choose $\epsilon < 0.01$; however, since the test required a sample size of $S > \sqrt{N}/\epsilon_2$, our sample size should be (at least) $2.56 \cdot 10^6$. However, this violates the other requirement: $S < N$ (since $N = 2^{16} < 2^{16} = N$).

Given the above limitations, we consider it interesting to observe how our four distributions will react under the Paninski test when using a sample size of $S = 60,000 < 2^{16} = N$.

For each of the four distributions, we would like to count how many bins have exactly one sample ($K_1$) and check if that is consistent with the obtained results in Chapter 4. Specifically, as we saw previously (in Chapter 4), among the four distributions, $(Z_{100}, Z_{101})$ is closest to the uniform distribution (whereas $(TK_1, TK_2)$ is the furthest). Thus, we expect that, for the distribution $(Z_{100}, Z_{101})$, we would obtain a larger $K_1$ than the case where the samples are drawn from $(TK_1, TK_2)$.

For that, for each of the four distributions, we draw 60,000 samples (randomly and independently; see the code in Appendixes B.2.1 and B.2.2). In addition, we count how many samples appeared exactly once in the sample (i.e., did not collide with any other sample). We repeat this experiment 500 times (to achieve better confidence in the results since using a relatively small sample (only 60,000 samples) may result in greater "noise"). At the end of each experiment, we save the results to a file.

Then, we run a Matlab script (see the code in Appendix B.2.3) that reads the data from the file and outputs the average value and the standard deviation of $K_1$. In Table 5.1, we summarize the results for the four distributions (including the program’s execution time).

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg($K_1$)</th>
<th>Std($K_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(TK_1, TK_2)$</td>
<td>1m.23s</td>
<td>23,846</td>
<td>126</td>
</tr>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>1m.21s</td>
<td>23,989</td>
<td>128</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>1m.29s</td>
<td>24,017</td>
<td>116</td>
</tr>
<tr>
<td>$(Z_{100}, Z_{101})$</td>
<td>1m.42s</td>
<td>24,019</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 5.1: Paninski test results, using a sample size of 60,000

As you can see, (on average) the further a distribution is from the uniform distribution (such as $(TK_1, TK_2)$), the fewer samples that did not collide at all.

For reference, we calculate the expectation of $K_1$ under the uniform distribution. We define $N$ Bernoulli random variables, $\chi_0, \chi_1, \ldots, \chi_{N-1}$, as follows: $\forall i \in \{0, 1, \ldots, N - 1\}$, $\chi_i = 1$, if (when using a sample size of $S$), in the $ith$ bin, there is exactly one sample (otherwise, $\chi_i = 0$).
We denote by $p_i$ the probability that a sample falls into the $i$th bin; thus:
\[
\Pr(\chi_i = 1) = p_i \cdot (1 - p_i)^{S - 1} \cdot \binom{S}{1}
\]
Since $\chi_i$ is a Bernoulli random variable, the following holds: $\mathbb{E}(\chi_i) = \Pr(\chi_i = 1)$. In addition, since $K_1$ counts how many bins have exactly one sample in them, then $K_1 = \sum_{i=0}^{N-1} \chi_i$, and due to the linearity of expectation, we obtain the following:
\[
\mathbb{E}(K_1) = \sum_{i=0}^{N-1} \mathbb{E}(\chi_i) = \sum_{i=0}^{N-1} p_i \cdot (1 - p_i)^{S - 1} \cdot \binom{S}{1}
\]
In the uniform distribution, $\forall i \in [N]$, $p_i = \frac{1}{N}$, and therefore,
\[
\mathbb{E}_U(K_1) = S \cdot \sum_{i=0}^{N-1} \frac{1}{N} \cdot \left(\frac{N - 1}{N}\right)^{S - 1} = S \cdot \left(\frac{N - 1}{N}\right)^{S - 1}
\]
Now, if we set $S = 60,000$ and $N = 2^{16}$ we obtain $\mathbb{E}_U(K_1) \approx 24,019$.

As you can see, this value is very close to the average values obtained for $(Z_{100}, Z_{101})$ and $(TK_{32}, TK_{33})$. Unfortunately, notice that the standard deviation of all the experiments is greater than the differences between the averages obtained. Therefore, we cannot use this test for distinguishing between the distributions. However, for the distribution $(TK_1, TK_2)$ we indeed obtained clearer results (the difference between its $K_1$ average and $\mathbb{E}_U(K_1) = 24,019$ is greater than one standard deviation), but it remains insufficient\footnote{Recall that its $L_1$ distance from the uniform distribution is approximately 0.06; see Section 4.2.}.

\footnote{This is not surprising since we used a sample whose size is smaller than the lower bound value: $\Omega \left( \frac{\sqrt{N}}{N} \right)$}
Chapter 6

The collision tester

The idea behind this method is to count the number of colliding pairs in the sample, i.e., to count how many pairs have the same value. More formally, suppose that we drew $S$ samples, $x_1, x_2, \ldots, x_S$, and let $i, j \in [1..S]$; $s.t \ i < j$ be two different indexes. Then, we say that $i$ and $j$ collide if $x_i = x_j$. Thus, the number of pairs that collide in the sample (denoted by $C_D$) is:

$$C_D = |\{(i, j); 1 \leq i < j \leq S : x_i = x_j\}|$$

For a fixed sample size $S$, there are $\binom{S}{2}$ pairs (in total). Then, using $C_D$, we can obtain a good estimation for the collision probability (for a pair) of the tested distribution.

Another important observation is that, for the uniform distribution (over the domain $[N]$), the collision probability (for a pair), $C_U$, is equal to $\frac{1}{N}$. With increasing distance (in the $L_1$ sense) from this distribution, the collision probability will increase.

As a result, there is an algorithm that can distinguish (with high probability) between the uniform distribution and a distribution that is $\epsilon$ (or more) further away [GR00]. The algorithm takes $S = O\left(\frac{\sqrt{N}}{\epsilon^2}\right)$ samples (randomly and independently) and counts how many pairs collide in the sample (i.e., $C_D$).

The algorithm accepts (e.g., outputs "1") the argument that the distribution is the uniform distribution if $C_D$ is smaller than some threshold; otherwise, it rejects the argument (e.g., outputs "0").

We want to check if, when using the number of colliding pairs ($C_D$), we can distinguish between our four tested distributions and whether the results will be consistent with the $L_1$ estimations obtained in Chapter 4. Specifically, we expect (for a sample with fixed size) that the distribution $(TK_1, TK_2)$ (i.e., the farthest distribution from the uniform distribution among our four tested distributions) creates the highest number of collisions, and the distribution $(Z_{100}, Z_{101})$ (which is the closest distribution among the four distributions) creates the lowest number of collisions.

\[\text{Recently, a new analysis [Dia+16] showed that } S = O\left(\frac{\sqrt{N}}{\epsilon^2}\right) \text{ is also sufficient. Recall footnote}\]
For this purpose, we wrote a C++ program (see the code in Appendixes B.3.1
and B.3.2) that runs 100 simulations.
In each simulation, we draw $S = 2^{\exp}$ samples from the tested distribution ("exp" is a parameter that is given to the program at the start and has to be in the range $15 \leq \exp \leq 40$).
During each simulation, the program counts the number of pairs that collide in the
sample, and it saves this value in a file (to obtain this value, we visit each domain
element and count how many pairs fall on it).

Then, we run a Matlab script (see the code in Appendix B.3.4) that reads the
values from the file and presents the scattering of the number of collisions as nor-
malized histograms (for reference, we also include the normal distribution obtained
when using the same expectation and standard deviation).
In the next section, we present the obtained results.

6.1 The simulation results

6.1.1 Using a sample size of $2^{18}$

In Figure 6.1, the (normalized) histograms of the number of colliding pairs are
presented when using a sample size of $2^{18}$ for each of our four distributions.

![Collision pairs tester using $2^{18}$ samples](image)

Figure 6.1: The number of colliding pairs for the four distributions, using a
sample size of $S = 2^{18}$

As you can see, the distribution $(TK_1, TK_2)$ (its histograms are presented in yellow) is very different from the other distributions. The number of colliding pairs (in this distribution) was the highest among the four measured distributions. This is consistent with the fact that $(TK_1, TK_2)$ is the furthest among our four distribu-
tions (relative to the uniform distribution), and its distance (from the uniform
distribution) is larger by almost an order of magnitude compared to the remaining measured distances (compare with the results in Section 4.2).
To achieve a better picture for the remaining three distributions, we zoom in near them:

Figure 6.2: The number of colliding pairs, using a sample size of $S = 2^{18}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$

As you can see, it is difficult (or even impossible) to distinguish between the remaining three distributions in this sample (based only on the collision tester results).

(One can notice that $(Z_1, Z_2)$ [its histograms are presented in blue] creates on average more collisions than do $(Z_{100}, Z_{101})$ [in green] and $(TK_{32}, TK_{33})$ [in red]).

In Table 6.1, we give the average and standard deviation of the number of colliding pairs, $C^*_D$, of the four tested distributions in this sample (its size is $2^{18}$). (In addition, we supply the program’s execution time).

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg($C^*_D$)</th>
<th>Std($C^*_D$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(TK_1, TK_2)$</td>
<td>1m.13s</td>
<td>611,140</td>
<td>2,498</td>
</tr>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>1m.11s</td>
<td>526,381</td>
<td>738</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>1m.19s</td>
<td>524,834</td>
<td>661</td>
</tr>
<tr>
<td>$(Z_{100}, Z_{101})$</td>
<td>1m.30s</td>
<td>524,280</td>
<td>746</td>
</tr>
</tbody>
</table>

Table 6.1: Collision tester results, using a sample size of $2^{18}$

Similar to the Paninski test results (see Chapter 5), one can see that the further a distribution is (from the uniform distribution), the more (pairs) collisions occurred in its sample (on average). However, as mentioned above, we cannot distinguish between the distributions (except for $(TK_1, TK_2)$) because the standard deviations are relatively high for the differences between the averages.
Therefore, in the next section, we continue to check with a larger sample.
6.1.2 Using a sample size of $2^{20}$

In Figure 6.3, the (normalized) histograms of the number of colliding pairs are presented, when using a sample size of $2^{20}$, for the following distributions: $(Z_1, Z_2)$ (in blue), $(TK_{32}, TK_{33})$ (red) and $(Z_{100}, Z_{101})$ (green).

![Histogram of colliding pairs](image)

Figure 6.3: The number of colliding pairs, using a sample size of $S = 2^{20}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$.

Now, you can see that $(Z_1, Z_2)$ is also distinguishable from the other distributions. The number of colliding pairs (in $(Z_1, Z_2)$) was larger (in all 100 simulations) than the number of colliding pairs in $(TK_{32}, TK_{33})$ and in $(Z_{100}, Z_{101})$. This indicates that the distance of $(Z_1, Z_2)$ from the uniform distribution is indeed greater than the distances of those distributions.

However, it remains difficult to distinguish between $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$ (those two distributions present the smallest distance from the uniform distribution among the four tested distributions). However, one can “feel” that the number of colliding pairs in $(TK_{32}, TK_{33})$ was larger than the number obtained in $(Z_{100}, Z_{101})$ (at least in terms of averages). 

\[\|D_{TK_{32}, TK_{33}, U_{2^{16}}} \|_1 > \|D_{Z_{100}, Z_{101}, U_{2^{16}}} \|_1.\]

\[11\] This may indicate that indeed $\|D_{TK_{32}, TK_{33}, U_{2^{16}}} \|_1 > \|D_{Z_{100}, Z_{101}, U_{2^{16}}} \|_1.$
As before (when $S = 2^{18}$) and as expected, the greatest number of collisions occurred in $(TK_1, TK_2)$.

In Figure 6.4, the scattering of the number of colliding pairs of $(TK_1, TK_2)$ (yellow) and $(Z_1, Z_2)$ (blue) is presented, when using a sample size of $2^{20}$.

![Collision pairs tester using $2^{20}$ samples](image)

**Figure 6.4:** The number of colliding pairs, using a sample size of $S = 2^{20}$, of the distributions $(Z_1, Z_2)$ and $(TK_1, TK_2)$

In Table 6.2, we give the average and standard deviation of the number of colliding pairs, $C_D$, of the four tested distributions in this sample (its size is $2^{20}$). (In addition, we supply the program’s execution time).

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg($C_D$)</th>
<th>Std($C_D$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(TK_1, TK_2)$</td>
<td>4m.55s</td>
<td>9,781,742</td>
<td>18,033</td>
</tr>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>4m.47s</td>
<td>8,421,666</td>
<td>2,991</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>5m.18s</td>
<td>8,394,518</td>
<td>2,946</td>
</tr>
<tr>
<td>$(Z_{100}, Z_{101})$</td>
<td>6m.2s</td>
<td>8,388,877</td>
<td>2,478</td>
</tr>
</tbody>
</table>

**Table 6.2:** Collision tester results, using a sample size of $2^{20}$

As you can see from the results, with a sample size of $S = 2^{20}$, it is easy to distinguish between $(TK_1, TK_2)$ and $(Z_1, Z_2)$ and the remaining distributions; however, it remains difficult to distinguish (clearly) between $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$. 
6.1.3 Using a sample size of $2^{22}$

In Figure 6.5, the (normalized) histograms of the number of colliding pairs are presented, when using a sample size of $2^{22}$, for the following distributions: $(Z_1, Z_2)$ (blue), $(TK_{32}, TK_{33})$ (red) and $(Z_{100}, Z_{101})$ (green).

Figure 6.5: The number of colliding pairs, using a sample size of $S = 2^{22}$, of the distributions $(Z_1, Z_2)$, $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$

Now, you can see that it is possible to distinguish (clearly) between all distributions. The obtained results are consistent with the relations of the distances between the distributions and the uniform distribution (as we learned in Chapter 4). The closest distribution to the uniform distribution (among our four tested distributions), $(Z_{100}, Z_{101})$, creates the lowest number of collisions.

In Table 6.3 we give the average and standard deviation of the number of colliding pairs, $C_D$, of the four tested distributions in this sample (its size is $2^{22}$). (In addition, we supply the program’s execution time).

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg($C_D$)</th>
<th>Std($C_D$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(TK_1, TK_2)$</td>
<td>20m.3s</td>
<td>156,504,668</td>
<td>143,000</td>
</tr>
<tr>
<td>$(Z_1, Z_2)$</td>
<td>19m.23s</td>
<td>134,746,477</td>
<td>14,867</td>
</tr>
<tr>
<td>$(TK_{32}, TK_{33})$</td>
<td>21m.10s</td>
<td>134,308,779</td>
<td>12,519</td>
</tr>
<tr>
<td>$(Z_{100}, Z_{101})$</td>
<td>24m.5s</td>
<td>134,216,846</td>
<td>11,905</td>
</tr>
</tbody>
</table>

Table 6.3: Collision tester results, using a sample size of $2^{22}$

Using a sample of only $2^{22}$ samples, and with an execution time of less than 25 minutes, we manage to obtain a good (qualitative) estimation of the relations of the distances between those distributions and the uniform distribution (over the same domain).

For a reference, recall that learning (directly) the $L_1$ distance (see Chapter 4) required $2^{38}$ samples and took us approximately 10 days.
6.1.4 Additional results

In addition to our four distributions, where their distances were learned using the learning algorithm in Chapter 4, we also would like to obtain a qualitative estimation (using the collision tester) of strong biases (relative to the uniform distribution) at additional locations in the RC4 keystream bytes (especially at the initial bytes).

For that, we also apply the collision tester using a sample size of $2^{28}$ on the distribution $(Z_4, Z_5)$ - (the pair consecutive bytes at locations 4 and 5 in the RC4 keystream). For reference, we compare the results with the distribution $(Z_{100}, Z_{101})$ and the "pseudo-uniform distribution" over the domain $[2^{16}] = \{0, 1, \ldots, 2^{16} - 1\}$. The pseudo-uniform distribution, which we denote by $\hat{U}$, referred to the distribution that was generated by invoking the random() function (from C++11) over the above domain (see Section 4.1 and the code in Appendix B.3.3).

In Figure 6.6, the (normalized) histograms of the number of colliding pairs are presented, using a sample size of $2^{28}$, for the above distributions: $(Z_4, Z_5)$ (purple), $(Z_{100}, Z_{101})$ (green) and $\hat{U}$ (black).

![Collision pairs tester using $2^{28}$ samples](image)

Figure 6.6: The number of colliding pairs, using a sample size of $S = 2^{28}$, of the distributions $(Z_4, Z_5)$, $(Z_{100}, Z_{101})$ and $\hat{U}$

In Table 6.4, we give the average and standard deviation of the number of colliding pairs, $C_D$, of the above three distributions, using a sample size of $2^{28}$. (In addition, we supply the program’s execution time\footnote{Notice that the execution time of $\hat{U}$ is substantially smaller than that of the other two distributions since, in $\hat{U}$, we do not use the KSA (for initializing the internal state) or the PRGA (for creating the keystream), in contrast to RC4 keystream byte distributions.}).


<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg($C_D$)</th>
<th>Std($C_D$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>($Z_4, Z_5$)</td>
<td>21h.6m.39s</td>
<td>5.4975647 · 10^{11}</td>
<td>7.1627 · 10^{5}</td>
</tr>
<tr>
<td>($Z_{100}, Z_{101}$)</td>
<td>26h.19m.28s</td>
<td>5.4975599 · 10^{11}</td>
<td>8.2040 · 10^{5}</td>
</tr>
<tr>
<td>$U$</td>
<td>23m.14s</td>
<td>5.4975583 · 10^{11}</td>
<td>7.7068 · 10^{5}</td>
</tr>
</tbody>
</table>

Table 6.4: Collision tester results, using a sample size of $2^{28}$

As you can see, it is impossible to distinguish between the above distributions using the collision tester even with relatively large samples ($S = 2^{28}$). This may indicate that the $L_1$ distance between ($Z_4, Z_5$) and the uniform distribution is smaller than the distance that we measured for ($TK_{32}, TK_{33}$) (see Table 4.2).

In addition, notice that we also obtain similar results as for the ”pseudo-uniform distribution”, $U$, which strengthens the sense that both distributions, ($Z_4, Z_5$) and ($Z_{100}, Z_{101}$), are indeed close to the uniform distribution.

Regardless, to attempt to distinguish between ($Z_4, Z_5$) and ($Z_{100}, Z_{101}$), we increase the sample size to $2^{32}$; this time, we are content with 20 simulations.

In Figure 6.7, the (normalized) histograms of the number of colliding pairs are presented, when using a sample size of $2^{32}$, for ($Z_4, Z_5$) (purple), ($Z_{100}, Z_{101}$) (green) and $U$ (black).

![Collision pairs tester using $2^{32}$ samples](image)

**Figure 6.7:** The number of colliding pairs, using a sample size of $S = 2^{32}$, of the distributions ($Z_4, Z_5$), ($Z_{100}, Z_{101}$) and $U$.

---

13 Recall that we have managed to distinguish between ($TK_{32}, TK_{33}$) and ($Z_{100}, Z_{101}$) even when using a sample size of $2^{22}$ (see the results from the previous section).

14 Close in the sense that they are indistinguishable (using the collision tester) even when using a sample size of $2^{28}$.
In Table 6.5, we give the average and standard deviation of the number of colliding pairs, \( \tilde{C}_D \), of the above three distributions, using a sample size of \( 2^{32} \). (In addition, we supply the program’s execution time\(^{15}\).

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Execution time</th>
<th>Avg(( \tilde{C}_D ))</th>
<th>Std(( \tilde{C}_D ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((Z_4, Z_5))</td>
<td>66h.18m.48s</td>
<td>1.40737718 \cdot 10^{14}</td>
<td>1.1891 \cdot 10^7</td>
</tr>
<tr>
<td>((Z_{100}, Z_{101}))</td>
<td>82h.29m.57s</td>
<td>1.40737541 \cdot 10^{14}</td>
<td>1.3505 \cdot 10^7</td>
</tr>
<tr>
<td>(\hat{U})</td>
<td>1h.14m.13s</td>
<td>1.40737492 \cdot 10^{14}</td>
<td>1.5335 \cdot 10^7</td>
</tr>
</tbody>
</table>

Table 6.5: Collision tester results, using a sample size of \( 2^{32} \)

Now, as you can see, it is possible to distinguish (clearly) between the distributions; \((Z_4, Z_5)\) creates the highest number of collisions, followed by \((Z_{100}, Z_{101})\), and \(\hat{U}\) creates the lowest number of collisions.

Now, we can conclude that the \( L_1 \) distance between \((Z_4, Z_5)\) and the uniform distribution is greater than the \( L_1 \) distance between \((Z_{100}, Z_{101})\) and the uniform distribution (namely, \( \| (Z_4, Z_5), U_{2^{16}} \|_1 > \| (Z_{100}, Z_{101}), U_{2^{16}} \|_1 \)).

\(^{15}\)See footnote \(^{12}\) above.
Chapter 7

Comparing the fingerprint

As we saw in the previous chapter, when using a sample size of $2^{20}$ we can distinguish between $(Z_1, Z_2)$, $(TK_1, TK_2)$ and the others distributions, using the collision tester, but it is hard to distinguish (clearly) between $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$. In addition, we saw that when using a sample size of $2^{22}$ it is possible to distinguish between all our four distributions, using the collision tester.

As mentioned in Section 2.6, the fingerprint of a sample contains all the information (collision statistics) that required for testing symmetric properties of a distribution (such as, its $L_1$ distance from the uniform distribution). This information contains in fact the number of domain elements that experience $k$-way collisions, for each $1 \leq k \leq S$. In particular, this information contains the number of colliding pairs in the sample, as the following holds:

$$C_D = \sum_{j=2}^{S} F(j) \cdot \binom{j}{2}$$

Since using the fingerprint may give us additional info (e.g., the number of colliding triplets), we found it interesting to check whether, when using a sample size of $2^{21}$, we can (already) distinguish (clearly) between all four distributions by comparing their fingerprints (in particular, distinguish between $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$).

To check this, we wrote a C++ program (see the code in Appendix B.4.1 and B.4.2) that runs 100 simulations. In each simulation, we draw $2^{21}$ samples of the keystream bytes (according to the tested distribution), and we write the obtained fingerprint to a file, that is, $F_D = (F_1, F_2, \ldots, F_{999})$. At the end of the program, we run a Matlab script (see code in Appendix B.4.3), that averages the 100 obtained fingerprints (from the 100 simulations) and graphically presents the average fingerprint: $\overline{F_D} = (\overline{F_1}, \overline{F_2}, \ldots, \overline{F_{999}})$. In addition, for reference, we add the expectation of the fingerprint obtained subject to the uniform distribution, $\overline{F_U}$.

We calculate this value as follows:
For each $1 \leq j \leq S$, we define $2^{16}$ Bernoulli random variables, $\chi^j_0, \chi^j_1, \ldots, \chi^{2^{16}-1}_j$, as follows:

$\chi^j_i = 1$ $(0 \leq i \leq 2^{16} - 1)$, if into the $i$th bin fell exactly $j$ samples (otherwise, $\chi^j_i = 0$).

The following holds:

$$\mathbb{E}(\chi^j_i) = \Pr(\chi^j_i = 1) = \binom{S}{j} \cdot p(i)^j \cdot (1 - p(i))^{S-j}$$

where $p(i)$ is the probability for a single sample to fall into the $i$th bin.

Since we refer to the uniform distribution, $\forall i \in [2^{16}]$, $p(i) = 2^{-16}$, and thus,

$$\mathbb{E}(\chi^j_i) = \binom{S}{j} \cdot (2^{-16})^j \cdot (1 - 2^{-16})^{S-j}$$

In addition, note that $F(j) = \sum_{i \in [2^{16}]} \chi^j_i$, and by using the linearity of expectation:

$$\overline{F_U}(j) = \sum_{i \in [2^{16}]} \mathbb{E}(\chi^j_i) = \sum_{i=0}^{2^{16}-1} \binom{S}{j} \cdot (2^{-16})^j \cdot (1 - 2^{-16})^{S-j} = 2^{16} \cdot \binom{S}{j} \cdot (2^{-16})^j \cdot (1 - 2^{-16})^{S-j}$$

Therefore, the expected value of the $j$th entry of the fingerprint (obtained subject to the uniform distribution) can be calculated using the last equation above.

Now, we present and analyze the obtained results.

In the following figures (Figures 7.1-7.8), the fingerprints of our four tested distributions are presented:

1. $(TK_1, TK_2)$ is presented in yellow.
2. $(Z_1, Z_2)$ is presented in blue.
3. $(TK_{32}, TK_{33})$ is presented in red.
4. $(Z_{100}, Z_{101})$ is presented in green.

(We choose the same mapping of colors to the distributions that we used in Chapter 6).

In addition, for reference, we present in black the expectation of the fingerprint obtained subject to the uniform distribution ($\overline{F_U}$).

All the results were obtained when using a sample size of $S = 2^{21}$.  

35
In Figure 7.1, entries 15-55 of the obtained average fingerprint vectors, namely, $F_D(15), F_D(16), \ldots, F_D(55)$, are presented.

![Fingerprint, using $N=2^{16}$ (number of domain elements) and $S=2^{21}$ (sample size)](image)

**Figure 7.1**: The average fingerprints (entries 15-55) of our four distributions

In the following figures (Figures 7.2-7.8), we zoom in around the fingerprint's entries to achieve a clearer picture.

In Figure 7.2, entries 14-21 of the obtained average fingerprints, namely, $F_D(14), F_D(15), \ldots, F_D(21)$, are presented.

![Fingerprint, using $N=2^{16}$ (number of domain elements) and $S=2^{21}$ (sample size)](image)

**Figure 7.2**: The average fingerprints (entries 14-21) of our four distributions
In Figure 7.3, entries 22-27 of the obtained average fingerprints, namely, $F_D(22), F_D(23), \ldots, F_D(27)$, are presented.

![Figure 7.3: The average fingerprints (entries 22-27) of our four distributions](image)

In Figure 7.4, entries 28-36 of the obtained average fingerprints, namely, $F_D(28), F_D(29), \ldots, F_D(36)$, are presented.

![Figure 7.4: The average fingerprints (entries 28-36) of our four distributions](image)
In Figure 7.5, entries 37-46 of the obtained average fingerprints, namely, $F_{D}(37), F_{D}(38), \ldots, F_{D}(46)$, are presented.

Figure 7.5: The average fingerprints (entries 37-46) of our four distributions

In Figure 7.6, entries 47-56 of the obtained average fingerprints, namely, $F_{D}(47), F_{D}(48), \ldots, F_{D}(56)$, are presented.

Figure 7.6: The average fingerprints (entries 47-56) of our four distributions
In Figure 7.7, entries 57-84 of the obtained average fingerprints, namely, $F_D(57), F_D(58), \ldots, F_D(84)$, are presented.

![Figure 7.7: The average fingerprints (entries 57-84) of our four distributions](image)

In these entries (57-84), it is easy to see the difference between the fingerprint of $(Z_1, Z_2)$ and the fingerprints of the other distributions. Notice that $F_U(i) = 0$, $i \geq 65$; on the other hand, $F_{(Z_1, Z_2)}(i) \neq 0$, when $65 \leq i \leq 84$. This phenomenon is probably due to the strong bias in the second byte of the keystream.

Recall that $\Pr(Z_2 = 0) \approx \frac{2}{256}$ (see also Section 2.2).
In Figure 7.8, entries 210-270 of the obtained (average) fingerprint vectors, namely, $F_D(210), F_D(211), \ldots, F_D(270)$, are presented.

Figure 7.8: The average fingerprints (entries 210-270) of our four distributions

These entries (210-270) are very far from the entries in which $F_U(i) \neq 0$ (i.e., when $i \leq 63$).

This (extraordinary) phenomenon occurs in $(TK_1, TK_2)$ and is apparently due to the very strong bias (which we found empirically) in the first TKIP keystream byte (when $(TSC_0, TSC_1) = (0x00, 0xFF)$; $Pr(TK_1 = 128) \approx \frac{7.5}{256}$.

As you can see from the obtained results, the fingerprint of $(TK_1, TK_2)$ is very different from the other fingerprints (and in particular from that of the uniform distribution), and thus, it can be inferred that $(TK_1, TK_2)$ is the farthest distribution from the uniform distribution (among our four distributions).

In addition, you can see that the fingerprint of $(Z_1, Z_2)$ is also different from the expectation of the fingerprint of the uniform distribution.

On the other hand, the fingerprints of $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$ are close to the expectation of the fingerprint of the uniform distribution.

Only in entries 48-53 is the difference between the fingerprint of $(TK_{32}, TK_{33})$ and the fingerprint of $(Z_{100}, Z_{101})$ larger than the standard deviation of the results. Specifically, (only) when $48 \leq i \leq 53$,

$$|\overline{F_{(TK_{32}, TK_{33})}}(i) - \overline{F_{(Z_{100}, Z_{101})}}(i)| \geq STD\left(\overline{F_{(TK_{32}, TK_{33})}}(i)\right)$$

(where $STD$ denotes the standard deviation).

From this, we conclude that it remains difficult (even when using all the information from the fingerprints) to distinguish between $(TK_{32}, TK_{33})$ and $(Z_{100}, Z_{101})$ when only using a sample size of $2^{21}$.

Specifically, we did not gain more than what we already obtained via the collision
tester (see Chapter 6). This may indicate that, regarding the property of estimating the distance between some distribution and the uniform distribution, most of the information is concentrated in the number of colliding pairs in the sample.
Chapter 8

Conclusion

When learning (empirically) the $L_1$ distance between some distribution and the uniform distribution, over the same domain (with size $N$), up to $\epsilon$ accuracy, thus requiring (theoretically) $O\left(\frac{N}{\epsilon^2}\right)$ samples, it is possible to test uniformity (namely, to determine with high probability whether a given distribution is the uniform distribution or is $\epsilon$ further away from it) using a sample size of $O\left(\frac{\sqrt{N}}{\epsilon}\right)$. Notice that the greater the number of domain elements ($N$) is, the more useful the testing algorithm will be relative to the direct learning algorithm (by a factor of $\sqrt{N}$).

As a result, the main benefit of the uniformity testing algorithms (and specifically the collision tester) is manifested when applied over big data.

In practice via the collision tester, we managed to distinguish between four distributions of pairs of consecutive RC4 keystream bytes ($N = 2^{16}$) (with different levels of distances from the uniform distribution) when using a sample size of $2^{22}$. For reference, recall that learning their $L_1$ distances using the (naive) learning algorithm required $2^{38}$ samples. After we checked the quality of the collision tester as a tool for estimating closeness to the uniform distribution on pairs of consecutive RC4 keystream bytes ($N = 2^{16}$), we suggest a few directions for future work as follows:

First, it is possible to improve the execution time (for both the learning and testing algorithms) using distributed infrastructure (e.g., using a distributed network; see Section 4.3), for example, using several processors in parallel for generating the samples. However, although this method will improve the execution time, it will not reduce the (total) sample size required.

In addition, the collision tester may also be applied to triplets ($N = 2^{24}$), quartets ($N = 2^{22}$), etc. of RC4 keystream bytes (at different locations in the stream). Over such domains, it is difficult (or impossible) to use the learning algorithm (since a huge sample is required); however, with the collision tester (or the Paninski test), the results can be obtained in a reasonable amount of time.

Finally, we emphasize that the collision tester is a generic tool for testing uniformity (not only in the RC4 context). Therefore, it is possible (and recommended) to also apply this test to other applications that require (effective) uniformity testing, for example, testing the quality of a pseudo-random number generator (PRNG) and in particular other cryptographically secure PRNGs.
Bibliography


Appendix A

An upper bound for the sample size

We will now show that, when using a sample size of $S = O \left( \frac{N}{\epsilon^2} \right)$ (where $N$ is the domain size), with high probability, the $L_1$ distance between the tested distribution (denoted by $p$) and the uniform distribution over the same domain (denoted by $U_N$) can be learned up to $\epsilon$ accuracy.

First notice that if $\|p, \hat{p}\|_1 \leq \epsilon$, then, due to the triangle inequality, the following holds:

$$\max (0, (\|\hat{p}, U_N\|_1 - \epsilon)) \leq \|p, U_N\|_1 \leq \|\hat{p}, U_N\|_1 + \epsilon$$

Thus, it is sufficient to show that, when using $S = O \left( \frac{N}{\epsilon^2} \right)$, $\|p, \hat{p}\|_1 \leq \epsilon$ with high probability (Theorem 1 below).

Then, we will show that you can reduce the error probability (denoted by $\delta$) by repeating the learning experiment $O \left( \log \frac{1}{\delta} \right)$ times and choosing the median value among the results (Theorem 2 below).

Theorem 1. For a sample size of $S = \frac{8N}{\epsilon^2}$, the following holds: $\|p, \hat{p}\|_1 \leq \epsilon$ with probability $1 - \frac{1}{\sqrt{8}} \approx 0.65$ (at least).

Proof. For each $i \in [N]$, denote by $X_i$ the ratio between the number of appearances of $i$ in the sample to the total number of samples ($S$) (i.e., $X_i = \hat{p}(i)$).

Define $Y_i \triangleq X_i \cdot S$ and notice that $Y_i \sim Bin(S, p(i))$ (as $Y_i$ counts the number of appearances of $i$ in a sample of size $S$).

Therefore,

$$\mathbb{E}(Y_i) = S \cdot p(i) \Rightarrow \mathbb{E}(X_i) = p(i) \quad (A.1)$$

$$\mathbb{V}(Y_i) = S \cdot p(i) \cdot (1 - p(i)) \Rightarrow \mathbb{V}(X_i) = \frac{p(i) \cdot (1 - p(i))}{S} \leq \frac{p(i)}{S} \quad (A.2)$$

$$\mathbb{E}(|X_i - p(i)|) \leq \mathbb{E} \left( |X_i - p(i)|^2 \right)^{1/2} = \mathbb{V}(X_i)^{1/2} \quad (A.3)$$

$$\mathbb{E} \left( \sum_{i \in [N]} |X_i - p(i)| \right) = \sum_{i \in [N]} \mathbb{E}(|X_i - p(i)|) \leq \sum_{i \in [N]} \sqrt{\frac{p(i)}{S} \leq \sqrt{\frac{N}{S}}} \quad (A.4)$$

In (A.3), the inequality is due to $\mathbb{V}(Z) = \mathbb{E}(Z^2) - \mathbb{E}^2(Z) \geq 0$, and the equality follows from (A.1).
In (A.4), the equality follows from the linearity of expectation; the first inequality is due to (A.3) and (A.2), and the last inequality is due to \( \sum_{i \in [N]} \sqrt{p(i)} \leq \sqrt{N} \) (which one can show using Lagrange multipliers).

From (A.4), when setting \( S = \frac{8N}{\epsilon^2} \), we obtain \( \mathbb{E} \left( \sum_{i \in [N]} |X_i - p(i)| \right) \leq \frac{\epsilon}{\sqrt{8}} \), which implies (by Markov’s inequality) that

\[
\Pr \left( \|p, \hat{p}\|_1 \geq \epsilon \right) = \Pr \left( \sum_{i \in [N]} |X_i - p(i)| \geq \epsilon \right) \leq \frac{1}{\sqrt{8}}
\]

Now, by applying the complementary event, the proof is complete.

We will next show that, if we repeat the learning experiment \( r = O \left( \log \frac{1}{\delta} \right) \) times and choose the median value from the results, then we achieve a good approximation for \( \|p, U_N\|_1 \) with probability (at least) \( 1 - \delta \).

We will formalize this in the next theorem.

First, let us define some notations:

For each experiment \( m \in \{1, 2, \ldots, r\} \), we denote by \( \hat{p}_m \) the empirical distribution (derived using the samples drawn in the \( m \)th experiment) and by \( \hat{d}_m \) its \( L_1 \) distance from the uniform distribution (i.e., \( \hat{d}_m \equiv \|\hat{p}_m, U_N\|_1 \)).

In addition, we denote by \( \hat{d} \) the median value among the \( r \) distances obtained (i.e., \( \hat{d} = \text{med}(\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_r) \)).

**Theorem 2.** If we repeat (independently) the learning experiment \( r = O \left( \log \frac{1}{\delta} \right) \) times (specifically, in each experiment, \( m \in \{1, 2, \ldots, r\} \), we use a sample size of \( S = \frac{8N}{\epsilon^2} \) and calculate \( \hat{d}_m \)), then the following holds with probability (at least) \( 1 - \delta \):

\[
\max(0, (\|p, U_N\|_1 - \epsilon)) \leq \hat{d} \leq \|p, U_N\|_1 + \epsilon
\]

Equivalently,

\[
\max(0, (\hat{d} - \epsilon)) \leq \|p, U_N\|_1 \leq \hat{d} + \epsilon
\]

**Proof.** Define \( r \) Bernoulli random (independent) variables, \( \chi_1, \chi_2, \ldots, \chi_r \), as follows:

\[
\chi_m = 1, \text{ if at the } m \text{th experiment } (1 \leq m \leq r) \text{ the following held:}
\]

\[
\max(0, (\|p, U_N\|_1 - \epsilon)) \leq \hat{d}_m \leq \|p, U_N\|_1 + \epsilon
\]

(A.5)

and \( \chi_m = 0 \) otherwise.

Define \( p_g \) as the probability of the (good) event, where \( \chi_m = 1 \), (i.e., \( p_g \equiv \Pr(\chi_m = 1) \)).

Since \( \chi_m \) is a Bernoulli variable, and from Theorem 1, its holds that

\[
\mathbb{E}(\chi_m) = p_g \geq 1 - \frac{1}{\sqrt{8}} \approx 0.65
\]

Define \( \chi = \sum_{m=1}^{r} \chi_m \) (specifically, \( \chi \) represents how many times (A.5) held out of \( r \)).

Notice that, if \( \frac{r}{\epsilon} > \frac{1}{2} \), the following holds for the median value, \( \hat{d} \):

\[
\max(0, (\|p, U_N\|_1 - \epsilon)) \leq \hat{d} \leq \|p, U_N\|_1 + \epsilon
\]
Therefore, it is sufficient to show that $\frac{X}{r} > \frac{1}{2}$ with probability (at least) $1 - \delta$. We will show this by using the Chernoff additive bound:

$$\Pr \left( \frac{X}{r} \leq \frac{1}{2} \right) \leq \Pr \left( \frac{X}{r} < p_g - 0.1 \right) \leq e^{-2 \cdot 0.1^2 \cdot r}$$

To bound the above probability, we demand that $e^{-2 \cdot 0.1^2 \cdot r} < \delta$, and thus, $r > 50 \cdot \ln \left( \frac{\delta}{\delta} \right)$. Therefore, when $r = O \left( \log \frac{1}{\delta} \right)$, then $\frac{X}{r} > \frac{1}{2}$ with probability (at least) $1 - \delta$, and this completes the proof. ■
Appendix B

Source Code

B.1 Learning code

B.1.1 RC4_learn.cpp

/* rc4_learn.cpp

This program generates 2^exp samples of RC4 keystream bytes
at positions (z_first_location,z_first_location+1), and counts
for each bin i in \{0,1,...,2^{16}-1\} how many samples fell into it.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <random>
#define N (1 << 16) // 2^16
#define MAX_U64 0xffffffffffffffff // 2^64-1

/* Initializes random number generator */
std::random_device rd;
std::mt19937_64 mt(rd);
std::uniform_int_distribution<unsigned long long> dist(0, MAX_U64);

/* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array 'S' using the key 'key', which
in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i<=255; i++)
        S[i]=i;
    j=0;
    for (i=0; i<=255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap S[i] and S[j]
        Swap(S[i],S[j]);
    }

/* Initialize the key with 16 bytes random values */
inline void create_key(void* key)
{
    int i;
    unsigned long long* key_u LLP = (unsigned long long*) key;
    for (i=0; i < 2; i++)
        *(key_u LLP+i) = dist(mt);
}
/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name Exponent FirstByteLocation" where:
Name - the name of the execution file (rc4_learn)
Exponent - Log (on base 2) of the number of samples required.
Should be in the range [25,45]
FirstByteLocation - First byte location in the keystream where we
want to start counting the values for learning the distribution
at position (z_first_location,z_first_location+1).
Should be in the range [1,10,000].
*/

int CheckArguments(int argc, char* argv[],int& exp, int& first_byte_location)
{
    {
        fprintf(stderr,"Bad call.
Usage: Name Exponent FirstByteLocation.
\n\nreturn 1;
    }
    exp = atoi(argv[1]);
    if (exp < 25 || exp > 45)
    {
        fprintf(stderr,"Bad argument.
Exponent should be in "
            "the range [25,45]\n\nreturn 1;
    }
    first_byte_location = atoi(argv[2]);
    if (first_byte_location < 1 || first_byte_location > 10000)
    {
        fprintf(stderr,"Bad argument.
FirstByteLocation should be in "
            "the range [1,10000]\n\nreturn 1;
    }
    return 0;
}

int main(int argc, char* argv[])
{  
    unsigned int i,j,m;
    unsigned long long l;
    unsigned char S[256];
    unsigned char key_str[16];
    char file_name[60];
    FILE *fp;

    unsigned int counter[N] ={0};
    unsigned short temp;
    char* temp_p = (char*)&temp;
    int exp, first_byte_location;

    if (CheckArguments(argc,argv,exp,first_byte_location))
        return 1;
    
    sprintf(file_name,60,"learn_rc4_%d_%d_exp_%d.txt",first_byte_location,
        first_byte_location+1,exp);
    fp = fopen(file_name, "w");
    if (fp == NULL)
    {
        fprintf(stderr,"Error opening file!\n\nreturn 1;
    }
    // Generating 2^exp samples
    for (l=0; l < (unsigned long long) (1LL << exp); l++)
    {
        create_key(key_str);
        RC4_KSA(S,key_str);
        // Generating the keystream bytes
        for(i=0,j=0,m=0;m < (unsigned int) (first_byte_location+1);m++)
        {
            i = (i + 1) % 256;
            j = (j + S[i]) % 256;
            // swap s[i] and s[j]
            Swap(S[i],S[j]);
        }
    }
    return 0;
}
if (m==(unsigned int) (first_byte_location-1))
{
    temp = S[(S[i]+S[j]) % 256];
}
else if (m== (unsigned int) first_byte_location)
{
    *(temp_p+1)=S[(S[i]+S[j]) % 256];
}

// "temp" is the value at (z_first_location,z_first_location+1)
// The byte at (z_first_location+1) chosen as the MSB.
counter[temp]++;

// Print the result to a file
for(m=0; m < N; m++)
{
    fprintf(fp, "%d\t%d\n", m, counter[m]);
}

fclose(fp);
return 0;
B.1.2 TKIP_learn.cpp

/* TKIP_learn.cpp

This program generates $2^{\text{exp}}$ samples of RC4 TKIP keystream bytes at positions $(z_{\text{first location}}, z_{\text{first location}}+1)$, with TSC0,TSC1 are the two LSB of the TKIP sequence counter TSC, and counts for each bin $i$ in $(0,1,...,2^{16}-1)$ how many samples fell into it.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <random>

#define N (1 << 16) // $2^{16}$
#define MAX_U32 0xffffffff // $2^{32}-1$

/* Initializes random number generator */
std::random_device rd;
std::mt19937 mt(rd());
std::uniform_int_distribution<unsigned int> dist(0, MAX_U32);

/* Swap between the values ‘a’ and ‘b’ */
inline void Swap(unsigned char& a, unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i, j;
    for (i=0; i < 255; i++)
        S[i] = i;
    j=0;
    for (i=0; i < 255; i++)
    {
        j = (j + S[i] + key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i], S[j]);
    }
}

/* Initialize the key. The first 3 bytes of the key are always depend only on TSC0 and TSC1, which in our case stayed constants. The rest of the key (13 bytes) is assigned with random values */
inline void create_key(void* key)
{
    int i;
    unsigned int* key_uip = (unsigned int*) key;
    unsigned char* key_ucp = (unsigned char*) key;
    key_uip[3] = dist(mt) % 256;
    for (i=1; i < 4; i++)
        *(key_uip+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid. Properly call for this program should be in the following form:
   "Name Exponent FirstByteLocation TSC0 TSC1" where:
   Name - the name of the execution file (TKIP_learn)
   Exponent - Log (on base 2) of the number of samples required. Should be in the range [25,45]
   FirstByteLocation - First byte location in the keystream where we want to start counting the values for learning the distribution at position $(z_{\text{first location}}, z_{\text{first location}}+1)$. Should be in the range [1,10,000].
   TSC0, TSC1 - The least-significant bytes (LSB) of the TKIP sequence counter "TSC". Should be in the range [0,255].
```c
/*
int CheckArguments(int argc, char* argv[], int& exp, int& first_byte_location,
        int& tsc0, int& tsc1)
{
    {
        fprintf(stderr, "Bad call. Usage: Name Exponent FirstByteLocation TSC0 TSC1
        .\n")
        return 1;
    }
    exp = atoi(argv[1]);
    if (exp < 25 || exp > 45)
    {
        fprintf(stderr, "Bad argument. Exponent should be in "
            "the range [25,45]\n")
        return 1;
    }
    first_byte_location = atoi(argv[2]);
    if (first_byte_location < 1 || first_byte_location > 10000)
    {
        fprintf(stderr, "Bad argument. FirstByteLocation should be in "
            "the range [1,10000]\n")
        return 1;
    }
    tsc0 = atoi(argv[3]);
    tsc1 = atoi(argv[4]);
    if (tsc0 < 0 || tsc0 > 255 ||
        tsc1 < 0 || tsc1 > 255)
    {
        fprintf(stderr, "Bad argument. TSC value should be in "
            "the range [0,255]\n")
        return 1;
    }
    return 0;
}
int main(int argc, char* argv[])
{
    unsigned int i, j, m;
    unsigned long long l;
    unsigned char S[256];
    unsigned char key_str[16];
    char file_name[60];
    FILE *fp;
    unsigned int counter[N] = {0};
    unsigned short temp;
    char* temp_p = (char*) &temp;
    int exp, first_byte_location, tsc0, tsc1;
    if (CheckArguments(argc, argv, exp, first_byte_location, tsc0, tsc1))
        return 1;
    // Initializing the first 3 bytes of the key, using TSC0 and TSC1
    key_str[0] = (unsigned char) tsc1;
    key_str[1] = ((unsigned char) tsc1 | 0x20) & 0x7f;
    key_str[2] = (unsigned char) tsc0;
    snprintf(file_name, 60, "learn_tkip_%d_%d_exp_%d.txt", first_byte_location,
        first_byte_location + 1, exp);
    fp = fopen(file_name, "w");
    if (fp == NULL)
    {
        fprintf(stderr, "Error opening file!\n");
        return (1);
    }
    // Generating 2^exp samples
    for (l = 0; l < (unsigned long long) (1LL << exp); l++)
    {
        create_key(key_str);
        RC4_KSA(S, key_str);
        // Generating the keystream bytes
        for (i = 0, j = 0, m = 0; m < (unsigned int) (first_byte_location + 1); m++)
            i = (i + 1) % 256;
    }
*/
```
j = (j + S[i]) % 256;
// swap s[i] and s[j]
Swap(S[i], S[j]);

if (m==(unsigned int) (first_byte_location-1))
{
    temp = S[(S[i]+S[j]) % 256];
}
else if (m==(unsigned int) first_byte_location)
{
    *(temp_p+1)=S[(S[i]+S[j]) % 256];
}

// "temp" is the value at (z_first_location, z_first_location+1)
// The byte at (z_first_location+1) chosen as the MSB.
counter[temp]++;
clear all
clc
format long

%% This script reads the samples data from the output files,
%% and calculates the L1 distance between the empirical distribution
%% and the uniform distribution.
%%
%% The four empirical distributions that we measured are:
%% 1. RC4 keystream bytes at location (z1,z2) - using 2^38 samples
%% 2. RC4 keystream bytes at location (z100,z101) - using 2^37 samples
%% 3. RC4 TKIP keystream bytes at location (z1,z2) when TSC0=0x00 and TSC1=0
%%    xFF - using 2^37 samples
%% 4. RC4 TKIP keystream bytes at location (z32,z33) when TSC0=0x00 and TSC1=0
%%    x00 - using 2^38 samples
%%}

N=2^37;
Expected = N / 2^16;

N2=2^38;
Expected2 = N2 / 2^16;

fileID = fopen('exp1\learn_tkip_1_2_exp_36.txt','r');
a1 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

fileID = fopen('exp2\learn_tkip_1_2_exp_36.txt','r');
a2 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

a=a1+a2;

fileID = fopen('exp1\learn_tkip_32_33_exp_36.txt','r');
b1 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

fileID = fopen('exp2\learn_tkip_32_33_exp_36.txt','r');
b2 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

b=b1+b2+b3;

fileID = fopen('exp1\learn_rc4_100_101_exp_36.txt','r');
c1 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

fileID = fopen('exp2\learn_rc4_100_101_exp_36.txt','r');
c2 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

c=c1+c2;

fileID = fopen('exp1\learn_rc4_1_2_exp_36.txt','r');
d1 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

fileID = fopen('exp2\learn_rc4_1_2_exp_36.txt','r');
d2 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

fileID = fopen('exp3\learn_rc4_1_2_exp_37.txt','r');
d3 = fscanf(fileID,'%d	%d
',size);
fclose(fileID);

d= d1+d2+d3;
diff=abs(a(2,:)-Expected);
L1_TKIP_1_2=sum(diff)/N;
msg = sprintf('Estimated L1 between TKIP (z1,z2) [when TSC0=0x00,TSC1=0xFF] and the U.D: %f.',L1_TKIP_1_2);
disp(msg)

diff2=abs(b(2,:)-Expected2);
L1_TKIP_32_33=sum(diff2)/N2;
msg = sprintf('Estimated L1 between TKIP (z32,z33) [when TSC0=0x00,TSC1=0x00] and the U.D: %f.',L1_TKIP_32_33);
disp(msg)

diff3=abs(c(2,:)-Expected);
L1_RC4_100_101=sum(diff3)/N;
msg = sprintf('Estimated L1 between RC4 (z100,z101) and the U.D: %f.',L1_RC4_100_101);
disp(msg)

diff4=abs(d(2,:)-Expected2);
L1_RC4_1_2=sum(diff4)/N2;
msg = sprintf('Estimated L1 between RC4 (z1,z2) and the U.D: %f.',L1_RC4_1_2);
disp(msg)
B.2 Paninski test code

B.2.1 RC4_Paninski_test.cpp

/* rc4_paninski_test.cpp
This program runs 500 simulations.
In each simulation, it generates "num_of_samples" samples of RC4 keystream
bytes
at positions (z_first_location, z_first_location+1), and counts
how many bins have exactly one sample which fell in them,
i.e. how many samples (in the sample) didn't collided with (any) other sample

The results are saved to a file.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <random>
#define N (1 << 16) // 2^16
#define MAX_U64 0xffffffffffffffff // 2^64-1
#define NUM_OF_SIMULATIONS 500
/* Initializes random number generator */
std::random_device rd;
std::mt19937_64 mt(rd());
std::uniform_int_distribution<unsigned long long> dist(0, MAX_U64);
/* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}
/* Initialize the permutation in the array "S" using the key "key", which
in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i <=255; i++)
        S[i]=i;
    j=0;
    for (i=0; i<=255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
    }
}
/* Initialize the key with 16 bytes random values */
inline void create_key(void* key)
{
    int i;
    unsigned long long* key_ullp = (unsigned long long*) key;
    for (i=0; i < 2 ; i++)
    *(key_ullp+i) = dist(mt);
}
/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name NumOfSamples FirstByteLocation" where:
Name - the name of the execution file (rc4_paninski_test)
NumOfSamples - The number of samples required.
Should be in the range (256,65,536).
FirstByteLocation - First byte location in the keystream where we
want to start counting the values for learning the distribution
at position (z_first_location, z_first_location+1).
Should be in the range [1,10,000].
*/
int CheckArguments(int argc, char* argv[],int& num_of_samples, int&
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
    unsigned int i, j, m;
    unsigned char S[256];
    unsigned char key_str[16];
    char file_name[60];
    FILE *fp;
    unsigned char counter[N];
    unsigned short temp,K1;
    char* temp_p = (char*)&temp;
    int num_of_samples, first_byte_location;

    if (CheckArguments(argc,argv,num_of_samples,first_byte_location))
        return 1;

    sprintf(file_name,60,"paninski_rc4_%d_%d_samplesize_%d.txt",
            first_byte_location,
            first_byte_location+1,num_of_samples);

    fp = fopen(file_name, "w");
    if (fp == NULL)
        return (1);

    // Running 500 simulations
    for(int iter=0; iter < NUM_OF_SIMULATIONS; iter++)
    {
        // Initializing the counters to zero
        K1 = 0;
        memset(counter,0,sizeof(counter));

        // Generating "num_of_samples" samples
        for (int l=0; l < num_of_samples; l++)
        {
            create_key(key_str);
            RC4_KSA(S,key_str);

            // Generating the keystream bytes
            for(i=0,j=0,m=0;m < (unsigned int) (first_byte_location+1);m++)
            {
                if (m==0)
                    temp = S[i] + S[j];
                else if (m== (unsigned int) first_byte_location)
                    temp = S[(S[i]+S[j]) % 256];
                else if (m== (unsigned int) first_byte_location-1)
                    temp = S[(S[i]+S[j]) % 256];
                else
                    temp = S[(S[i]+S[j]) % 256];

                S[i] = S[j];
                S[j] = temp;
                m++;
            }
        }
    }

    return 0;
}

int CheckArguments(int argc, char* argv[], int& num_of_samples, int& first_byte_location)
{
    {
        fprintf(stderr,"Bad call.
Usage: Name NumOfSamples FirstByteLocation.
");
        return 1;
    }

    num_of_samples = atoi(argv[1]);
    if (num_of_samples < 256 || num_of_samples > 65536 )
    {
        fprintf(stderr,"Bad argument.
NumOfSamples should be in "
             "the range [256,65536]\n");
        return 1;
    }

    first_byte_location = atoi(argv[2]);
    if (first_byte_location < 1 || first_byte_location > 10000)
    {
        fprintf(stderr,"Bad argument.
FirstByteLocation should be in "
              "the range [1,10000]\n");
        return 1;
    }

    return 0;
}
*(temp_p+1)=S[(S[i]+S[j]) % 256];
}

// "temp" is the value at (z_first_location, z_first_location+1)
// The byte at (z_first_location+1) chosen as the MSB.
counter[temp]++;

// Counting how many "bins" have exactly 1 sample in them.
for (m=0; m < N; m++)
{
   if (counter[m] == 1)
      K1++;

// Print the result to a file
fprintf(fp, "%d\n", K1);
}

fclose(fp);
return 0;
B.2.2 TKIP_Paninski_test.cpp

/* TKIP_Paninski_test.cpp

This program runs 500 simulations. In each simulation, it generates "num_of_samples" samples of RC4 TKIP keystream bytes at positions (z_first_location,z_first_location+1), with TSC0,TSC1 are the two LSB of the TKIP sequence counter TSC, and counts how many bins have exactly one sample which fell in them, i.e. how many samples (in the sample) didn't collided with (any) other sample.

The results are saved to a file.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <random>

#define N (1 << 16) // 2^16
#define MAX_U32 0xffffffff // 2^32-1
#define NUM_OF_SIMULATIONS 500

/* Initializes random number generator */
std::random_device rd;
std::mt19937 mt(rd);
std::uniform_int_distribution<unsigned int> dist(0, MAX_U32);

//* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i <= 255; i++)
        S[i] = i;
    j = 0;
    for (i=0; i <= 255; i++)
    {
        j = (j + S[i] + key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i], S[j]);
    }
}

/* Initialize the key. The first 3 bytes of the key are always depend only on TSC0 and TSC1, which in our case stayed constants. The rest of the key (13 bytes) is assigned with random values */
inline void create_key(void* key)
{
    int i;
    unsigned int* key_uip = (unsigned int*) key;
    unsigned char* key_ucp = (unsigned char*) key;
    key_uip[0] = dist(mt) % 256;
    for (i=1; i < 4; i++)
        *(key_uip+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid. Properly call for this program should be in the following form:
   "Name NumOfSamples FirstByteLocation TSC0 TSC1" where:
   Name - The name of the execution file (TKIP_Paninski_test)
   NumOfSamples - The number of samples required.
   Should be in the range [256,65,536].
   FirstByteLocation - First byte location in the keystream where we want to start counting the values for learning the distribution

59
/ int CheckArguments(int argc, char* argv[], int& num_of_samples, int& first_byte_location, int& tsc0, int& tsc1)
{
  {
    fprintf(stderr, "Bad call. Usage: Name NumOfSamples FirstByteLocation TSC0 TSC1\n");
    return 1;
  }
  num_of_samples = atoi(argv[1]);
  if (num_of_samples < 256 || num_of_samples > 65536)
  {
    fprintf(stderr, "Bad argument. NumOfSamples should be in \"the range [256, 65536]\n");
    return 1;
  }
  first_byte_location = atoi(argv[2]);
  if (first_byte_location < 1 || first_byte_location > 10000)
  {
    fprintf(stderr, "Bad argument. FirstByteLocation should be in \"the range [1, 10000]\n");
    return 1;
  }
  tsc0 = atoi(argv[3]);
  tsc1 = atoi(argv[4]);
  if (tsc0 < 0 || tsc0 > 255 || tsc1 < 0 || tsc1 > 255)
  {
    fprintf(stderr, "Bad argument. TSC value should be in \"the range [0, 255]\n");
    return 1;
  }
  return 0;
}

int main(int argc, char* argv[])
{
  unsigned int i, j, m;
  unsigned char S[256];
  unsigned char key_str[16];
  char file_name[60];
  FILE *fp;

  unsigned char counter[N];
  unsigned short temp, K1;
  char* temp_p = (char*) &temp;
  int num_of_samples, first_byte_location, tsc0, tsc1;

  if (CheckArguments(argc, argv, num_of_samples, first_byte_location, tsc0, tsc1))
    return 1;

  // Initializing the first 3 bytes of the key, using TSC0 and TSC1
  key_str[0] = (unsigned char) tsc1;
  key_str[1] = ((unsigned char) tsc1 | 0x20) & 0x7f;
  key_str[2] = (unsigned char) tsc0;

  snprintf(file_name, 60, "paninski_tkip_%d_%d_samplesize_%d.txt", first_byte_location, first_byte_location + 1, num_of_samples);

  fp = fopen(file_name, "w");
  if (fp == NULL)
    {
      fprintf(stderr, "Error opening file!\n");
      return 1;
    }

  // Running 500 simulations
  for (int iter = 0; iter < NUM_OF_SIMULATIONS; iter++)
  

// Initializing the counters to zero
K1 = 0;
memset(counter, 0, sizeof(counter));

// Generating "num_of_samples" samples
for (int l = 0; l < num_of_samples; l++)
{
    create_key(key_str);
    RC4_KSA(S, key_str);

    // Generating the keystream bytes
    for (i = 0, j = 0, m = 0; m < (unsigned int) (first_byte_location + 1); m++)
    {
        i = (i + 1) % 256;
        j = (j + S[i]) % 256;
        // swap s[i] and s[j]
        Swap(S[i], S[j]);

        if (m == (unsigned int) (first_byte_location - 1))
        {
            temp = S[(S[i] + S[j]) % 256];
        }
        else if (m == (unsigned int) first_byte_location)
        {
            *(temp_p + 1) = S[(S[i] + S[j]) % 256];
        }
    }

    // "temp" is the value at (z_first_location, z_first_location+1)
    // The byte at (z_first_location+1) chosen as the MSB.
    counter[temp]++;
}

// Counting how many "bins" have exactly 1 sample in them.
for (m = 0; m < N; m++)
{
    if (counter[m] == 1)
        K1++;
}

// Print the result to a file
fprintf(fp, "%d\n", K1);

fclose(fp);
return 0;
% This script reads the K1 data from the output files, 
% and calculates the average and standard deviation of K1 
% in the empirical distributions.
%
% The four empirical distributions, using 500 simulations with 60,000 samples 
% each:
% 1. RC4 keystream bytes at location (z1,z2)
% 2. RC4 keystream bytes at location (z100,z101)
% 3. RC4 TKIP keystream bytes at location (z1,z2) when TSC0=0x00 and TSC1=0
% 4. RC4 TKIP keystream bytes at location (z32,z33) when TSC0=0x00 and TSC1=0
%
N=60000;
m=2^16;
% The expectation value in case of Uniform distribution, 
% for reference only.
Expec=round(N*((m-1)/m)^(N-1));

fileID = fopen('paninski_rc4_1_2_samplesize_60000.txt','r');
A = fscanf(fileID,'%d
');
close(fileID);
fileID = fopen('paninski_rc4_100_101_samplesize_60000.txt','r');
B = fscanf(fileID,'%d
');
close(fileID);
fileID = fopen('paninski_tkip_1_2_samplesize_60000.txt','r');
C = fscanf(fileID,'%d
');
close(fileID);
fileID = fopen('paninski_tkip_32_33_samplesize_60000.txt','r');
D = fscanf(fileID,'%d
');
close(fileID);

msg=sprintf('TKIP (z_1,z_2) (when TSC0=0x00,TSC1=0xFF), k1 average value: %d	STD: %d',round(mean(C)),round(std(C)));
disp(msg)
msg=sprintf('(z_1,z_2) k1 average value: %d	STD: %d',round(mean(A)),round(std(A)));
disp(msg)
msg=sprintf('TKIP (z_32,z_33) (when TSC0=0x00,TSC1=0x00), k1 average value: %d	STD: %d',round(mean(D)),round(std(D)));
disp(msg)
msg=sprintf('(z_100,z_101) k1 average value: %d	STD: %d',round(mean(B)),round(std(B)));
disp(msg)
msg=sprintf('Uniform distribution k1 expectation value: %d.',Expec);
disp(msg)
B.3 Collision tester code

B.3.1 RC4_collision_test.cpp

/* collision_rc4.cpp

This program runs 100 simulations.
In each simulation, it generates \(2^{\exp}\) samples of RC4 keystream bytes
at positions \((z_{\text{first location}}, z_{\text{first location}}+1)\), and counts
pairs "collisions": the number of times that samples coincidentally
fall on the same domain element (a bin).
The results are saved to a file.
*/

#include <string.h>
#include <stdlib.h>
#include <random>
#include <iostream>

#define N (1 << 16) // \(2^{16}\)
#define MAX_U64 0xffffffffffffffff // \(2^{64} - 1\)
#define NUM_OF_REPETITIONS 100

/* Initializes random number generator */
std::random_device rd;
std::mt19937_64 mt(rd);
std::uniform_int_distribution<unsigned long long> dist(0, MAX_U64);

/* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which
in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i<=255; i++)
        S[i]=i;
    j=0;
    for (i=0; i<=255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
    }
}

/* Initialize the key with 16 bytes random values */
inline void create_key(void* key)
{
    int i;
    unsigned long long* key ullamp = (unsigned long long*) key;
    for (i=0; i < 2; i++)
        *(key ullamp+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name Exponent FirstByteLocation" where:
Name - The name of the execution file (collision_rc4)
Exponent - Log (on base 2) of the number of samples required.
Should be in the range \([15,40]\)
FirstByteLocation - First byte location in the keystream where we
want to start counting the values for learning the distribution
at position \((z_{\text{first location}}, z_{\text{first location}}+1)\).
Should be in the range \([1,10,000]\). */
int CheckArguments(int argc, char* argv[],int& exp, int& first_byte_location)
  {
    fprintf(stderr,"Bad call.\nUsage: Name Exponent FirstByteLocation.\n")
    return 1;
  }
exp = atoi(argv[1]);
if (exp < 15 || exp > 40)
  {
    fprintf(stderr,"Bad argument.\nExponent should be in "
    "the range [15,40]\n")
    return 1;
  }
first_byte_location = atoi(argv[2]);
if (first_byte_location < 1 || first_byte_location > 10000)
  {
    fprintf(stderr,"Bad argument.\nFirstByteLocation should be in "
    "the range [1,10000]"
    );
    return 1;
  }
return 0;

int main(int argc, char* argv[])
{
  unsigned int i, j, m;
  unsigned long long l, sum;
  unsigned char S[256];
  unsigned char key_str[16];
  char file_name[60];
  FILE *fp = NULL;
  unsigned int counter[N];
  unsigned short temp;
  char* temp_p = (char*)&temp;
  int exp, first_byte_location;
  if (CheckArguments(argc, argv, exp, first_byte_location))
    return 1;
  sprintf(file_name,60,"collision_rc4_%d_%d_exp_%d.txt",first_byte_location,
    first_byte_location+1,exp);
  fp = fopen(file_name, "w");
  if (fp == NULL)
    {
      fprintf(stderr,"Error opening file!\n")
      return (1);
    }
  // Running 100 simulations
  for(unsigned int iter=0; iter < NUM_OF_REPETITIONS; iter++)
  {
    // Initializing the counters to zero
    memset(counter,0,sizeof(counter));
    // Generating 2^("exp") samples
    for (l=0; l < (unsigned long long) (1LL << exp); l++)
    {
      create_key(key_str);
      RC4_KSA(S,key_str);
      // Generating the keystream bytes
      for(i=0,j=0,m=0;m < (unsigned int) (first_byte_location+1);m++)
      {
        i = (i + 1) % 256;
        j = (j + S[i]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
        if (m== (unsigned int) (first_byte_location-1))
            {
              temp = S[(S[i]+S[j]) % 256];
            }
        else if (m== (unsigned int) first_byte_location)
            {
              *(temp_p+1)=S[(S[i]+S[j]) % 256];
            }
// "temp" is the value at (z_first_location,z_first_location+1)
// The byte at (z_first_location+1) chosen as the MSB.
counter[temp]++;

// Counting the "pairs collisions"
for(i = 0 ; i < N ; i++ )
{
    sum+=(unsigned long long) counter[i]*(counter[i]-1)/2;
}

// Printing the result to a file
fprintf(fp,"%llu\n",sum);

fclose(fp);
return 0;
B.3.2 TKIP

collision_TKIP.cpp

/* collision_TKIP.cpp

This program runs 100 simulations. In each simulation, it generates \(2^\text{"exp"}\) samples of RC4 TKIP keystream bytes at positions \((z\_first\_location, z\_first\_location+1)\), with TSC0, TSC1 are the two LSB of the TKIP sequence counter TSC, and counts pairs "collisions" : the number of times that samples coincidentally fall on the same domain element (a bin). The results are saved to a file.
*/

#include <string.h>
#include <stdlib.h>
#include <random>
#include <iostream>

#define N (1 << 16) // 2^16
#define MAX_U32 0xffffffff // 2^32-1
#define NUM_OF_REPETITIONS 100

/* Initializes random number generator */
std::random_device rd;
std::mt19937 mt(rd());
std::uniform_int_distribution<unsigned int> dist(0, MAX_U32);

/* Swap between the values ‘a’ and ‘b’ */
inline void Swap(unsigned char& a, unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i <=255; i++)
        S[i]=i;
    j=0;
    for (i=0; i<=255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
    }
}

/* Initialize the key. The first 3 bytes of the key are always depend only on TSC0 and TSC1, which in our case stayed constants. The rest of the key (13 bytes) is assigned with random values */
inline void create_key(void* key)
{
    int i;
    unsigned int* key_uip = (unsigned int*) key;
    unsigned char* key_ucp = (unsigned char*) key;
    key_ucp[3] = dist(mt) % 256;
    for (i=1; i < 4; i++)
        *(key_uip+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid. Properly call for this program should be in the following form: "Name Exponent FirstByteLocation TSC0 TSC1" where:
Name - the name of the execution file (collision_TKIP)
Exponent - Log (on base 2) of the number of samples required. Should be in the range [15,40]
FirstByteLocation - First byte location in the keystream where we want to start counting the values for learning the distribution at position \((z\_first\_location, z\_first\_location+1)\).
TSC0, TSC1 - The least-significant bytes (LSB) of the TKIP sequence counter "TSC".
Should be in the range [0,255].

TSC0, TSC1 - The least-significant bytes (LSB) of the TKIP sequence counter "TSC".
Should be in the range [0,255].

*/
int CheckArguments(int argc, char* argv[], int& exp, int& first_byte_location, int& tsc0, int& tsc1)
{
{
    fprintf(stderr, "Bad call.\nUsage: Name Exponent FirstByteLocation TSC0 TSC1\n\n");
    return 1;
}
exp = atoi(argv[1]);
if (exp < 15 || exp > 40)
{
    fprintf(stderr, "Bad argument.\nExponent should be in "
            "the range [15,40]\n");
    return 1;
}
first_byte_location = atoi(argv[2]);
if (first_byte_location < 1 || first_byte_location > 10000)
{
    fprintf(stderr, "Bad argument.\nFirstByteLocation should be in "
            "the range [1,10000]\n");
    return 1;
}
tsc0 = atoi(argv[3]);
tsc1 = atoi(argv[4]);
if (tsc0 < 0 || tsc0 > 255 ||
tsc1 < 0 || tsc1 > 255)
{
    fprintf(stderr, "Bad argument.\nTSC value should be in "
            "the range [0,255]\n");
    return 1;
}
return 0;
}

int main(int argc, char* argv[])
{
    unsigned int i,j,m;
    unsigned long long l,sum;
    unsigned char S[256];
    unsigned char key_str[16];
    char file_name[60];
    FILE *fp = NULL;

    unsigned int counter[N];
    unsigned short temp;
    char* temp_p = (char*)&temp;
    int exp, first_byte_location, tsc0, tsc1;

    if (CheckArguments(argc, argv, exp, first_byte_location, tsc0, tsc1))
        return 1;

    // Initializing the first 3 bytes of the key, using TSC0 and TSC1
    key_str[0] = (unsigned char) tsc1;
    key_str[1] = ((unsigned char)tsc1|0x20)&0x7f;
    key_str[2] = (unsigned char) tsc0;

    snprintf(file_name, 60, "collision_tkip_%d_%d_exp_%d.txt", first_byte_location,
            first_byte_location+1, exp);
    fp = fopen(file_name, "w");
    if (fp == NULL)
    {
        fprintf(stderr, "Error opening file!\n");
        return (1);
    }

    // Running 100 simulations
    for(unsigned int iter=0; iter < NUM_OF_REPETITIONS; iter++)
    {  

// Initializing the counters to zero
define sum=0;
define memset(counter,0,sizeof(counter));

// Generating 2^("exp") samples
for (l=0; l < (unsigned long long) (1LL << exp); l++)
{
define create_key(key_str);
define RC4_KSA(S,key_str);

// Generating the keystream bytes
for(i=0,j=0,m=0;m < (unsigned int) (first_byte_location+1);m++)
{
  i = (i + 1) % 256;
  j = (j + S[i]) % 256;
  // swap s[i] and s[j]
  Swap(S[i],S[j]);

  if (m== (unsigned int) (first_byte_location-1))
  {
    temp = S[(S[i]+S[j]) % 256];
  }
  else if (m== (unsigned int) first_byte_location)
  {
    *(temp_p+1)=S[(S[i]+S[j]) % 256];
  }

  // "temp" is the value at (z_first_location,z_first_location+1)
  // The byte at (z_first_location+1) chosen as the MSB.
  counter[temp]++;
}

// Counting the "pairs collisions"
for(i = 0 ; i < N ; i++ )
{
define sum+=(unsigned long long) counter[i]*(counter[i]-1)/2;

// Printing the result to a file
fprintf(fp,"%llu\n",sum);
}

fclose(fp);
return 0;
B.3.3 Random_collision_test.cpp

/* collision_random.cpp

This program runs 100 simulations.
In each simulation, it generates $2^\exp$ samples of random values in the
range $[0,2^{16}-1]$, and counts the pairs "collisions":
the number of times that samples coincidentally fall on
the same domain element (a bin).
The results are saved to a file.
*/

#include <string.h> // for memset()
#include <iostream>
#include <random>
#define N (1 << 16) // $2^{16}$
#define MAX_U16 0xFFFF // $2^{16}-1$
#define NUM_OF_REPETITIONS 100

/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name Exponent" where:
Name - The name of the execution file (collision_random)
Exponent - Log (on base 2) of the number of samples required.
Should be in the range [15,40]
*/

int CheckArguments(int argc, char* argv[],int& exp)
{
    if (argc !=2 || argv[1] == NULL )
    {fprintf(stderr,"Bad call.
Usage: Name Exponent.\n");
        return 1;
    }
    exp = atoi(argv[1]);
    if (exp < 15 || exp > 40)
    {fprintf(stderr,"Bad argument.
Exponent should be in "
        "the range [15,40]\n");
        return 1;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    unsigned int counter[N];
    unsigned short rand_value;
    unsigned long long i, sum;
    FILE * fp=NULL;
    int exp;
    char file_name[50];
    if (CheckArguments(argc,argv,exp))
            return 1;
    snprintf(file_name,50,"collision_test_random_%d.txt",exp);
    fp = fopen(file_name, "w");
    if (fp == NULL)
        {fprintf(stderr,"Error opening file!\n");
                return (1);
        }

    /* Intializes random number generator */
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<unsigned short> dist(0, MAX_U16);

    // Running 100 simulations
    for(unsigned int j=0; j< NUM_OF_REPETITIONS; j++)
    {
        // Initializing the counters to zero
        sum=0;
        memset(counter,0,sizeof(counter));
    }

70
// Generating $2^\text{exp}$ samples
for (i = 0 ; i < (unsigned long long) (1LL << exp) ; i++ )
{
    rand_value = dist(mt);
    counter[rand_value]++;
}

// Counting the "pairs collisions"
for (i = 0 ; i < N ; i++ )
{
    sum+= (unsigned long long) counter[i] * (counter[i] - 1) / 2;
}

// Printing the result to a file
fprintf(fp, "%llu\n", sum);
}
fclose(fp);
return(0);
clear all
clc
format long

% This script reads the numbers of pairs collision data from the output files,
% and draws histograms for the results (based on 100 simulations), as follow:
% on figure(1): (histograms of the number of pairs collision)
% using Sample size of 2^18.
% The measured distribution are:
% 1. RC4 keystream bytes at location (z1,z2)
% 2. RC4 keystream bytes at location (z100,z101)
% 3. RC4 TKIP keystream bytes at location (z1,z2) when
% TSC0=0x00 and TSC1=0xFF
% 4. RC4 TKIP keystream bytes at location (z32,z33) when
% TSC0=0x00 and TSC1=0x00

fileID = fopen('collision_tkip_32_33_exp_18.txt','r');
tkip_32_33_vec = fscanf(fileID,'%d
');
close(fileID);
figure(1)
[m,x]=hist(tkip_32_33_vec,12);
b1=bar(x,m/trapz(x,m),'r');
hold on
b = 522000:50:529000;
mu = mean(tkip_32_33_vec);
sigma = std(tkip_32_33_vec);
f = exp(-(b-mu).^2./(2.*sigma.^2))./(sigma*sqrt(2*pi));
p1=plot(b,f,{'r', 'LineWidth', 4});

fileID = fopen('collision_rc4_1_2_exp_18.txt','r');
rc4_1_2_vec = fscanf(fileID,'%d
');
close(fileID);
[m,x]=hist(rc4_1_2_vec,12);
b2=bar(x,m/trapz(x,m),'b');
mu = mean(rc4_1_2_vec);
sigma = std(rc4_1_2_vec);
f = exp(-(b-mu).^2./(2.*sigma.^2))./(sigma*sqrt(2*pi));
p2=plot(b,f,{'b', 'LineWidth', 4});

fileID = fopen('collision_rc4_100_101_exp_18.txt','r');
rc4_100_101_vec = fscanf(fileID,'%d
');
close(fileID);
[m,x]=hist(rc4_100_101_vec,12);
b3=bar(x,m/trapz(x,m),'g');
mu = mean(rc4_100_101_vec);
sigma = std(rc4_100_101_vec);
f = exp(-(b-mu).^2./(2.*sigma.^2))./(sigma*sqrt(2*pi));
p3=plot(b,f,{'Color', [0.3, 0.8, 0.3], 'LineWidth', 3});

fileID = fopen('collision_tkip_1_2_exp_18.txt','r');
tkip_1_2_vec = fscanf(fileID,'%d
');
close(fileID);
[m,x]=hist(tkip_1_2_vec,12);
b4=bar(x,m/trapz(x,m),'y');
b = 600000:50:620000;
mu = mean(tkip_1_2_vec);
sigma = std(tkip_1_2_vec);
f = exp(-((b-mu)^2)/(2*sigma^2))/(sigma*sqrt(2*pi));
p4=plot(b,f,'Color', [1, 1, 0], 'LineWidth', 3);

set(gca, 'FontSize', 20)
xlabel('Number of colliding pairs', 'FontSize', 20, 'FontWeight', 'bold');
ylabel('Normalized histogram', 'FontSize', 20, 'FontWeight', 'bold');
axis([510000 640000 0 0.001]);

l=legend([p2,p3,p1,p4], '(z_1,z_2)', '(z_{100},z_{101})', 'TKIP (z_{32},z_{33})', 'TKIP (z_1,z_2)');
v = get(l,'title');
LEG = findobj(l,'type','text');
set(LEG,'FontSize',18)
set(v,'string','Distributions', 'FontWeight', 'bold', 'FontSize', 20);
title('Collision pairs tester using 2^{18} samples', 'FontSize', 24);
grid on
hold off

% on figure(2): (histograms of the number of pairs collision)
% using Sample size of 2^{20}.
The measured distribution are:
% 1. RC4 keystream bytes at location (z1,z2)
% 2. RC4 keystream bytes at location (z100,z101)
% 3. RC4 TKIP keystream bytes at location (z32,z33) when TSC0=0
% x00 and TSC1=0x00

fileID = fopen('collision_tkip_32_33_exp_20.txt','r');
tkip_32_33_vec = fscanf(fileID,'%d
'); fclose(fileID);
figure(2)
[m,x]=hist(tkip_32_33_vec,12);
b1=bar(x,m/trapz(x,m),'r');
hold on
b = 8381000:200:8405000;
mu = mean(tkip_32_33_vec);
sigma = std(tkip_32_33_vec);
f = exp(-((b-mu)^2)/(2*sigma^2))/(sigma*sqrt(2*pi));
p1=plot(b,f,'r','LineWidth',4);

fileID = fopen('collision_rc4_1_2_exp_20.txt','r');
rc4_1_2_vec = fscanf(fileID,'%d
'); fclose(fileID);

[m,x]=hist(rc4_1_2_vec,12);
b2=bar(x,m/trapz(x,m), [b, f', 'Color', [1, 1, 0], 'LineWidth', 4); 

fileID = fopen('collision_rc4_100_101_exp_20.txt','r');
rc4_100_101_vec = fscanf(fileID,'%d
'); fclose(fileID);

[m,x]=hist(rc4_100_101_vec,12);
b3=bar(x,m/trapz(x,m), [b, f', 'Color', [1, 1, 0], 'LineWidth', 4);
\( b = 8380000:200:8400000; \)
\( \mu = \text{mean}(rc4_{100,101}\_vec); \)
\( \sigma = \text{std}(rc4_{100,101}\_vec); \)
\( f = \exp\left(-\frac{(b-\mu)^2}{2\sigma^2}\right)/\left(\sigma \sqrt{2\pi}\right); \)
\( p3=\text{plot}(b,f,\text{Color',[0.3, 0.8, 0.3]},\text{LineWidth',3}); \)
\set(gca, 'FontSize', 20)
\xlabel('Number of colliding pairs','FontSize',20,'FontWeight','bold');
\ylabel('Normalized histogram','FontSize',20,'FontWeight','bold');
\l=\text{legend}([p2,p3,p1],'(z_1,z_2)','$TKIP (z_{32},z_{33})$');
\v = \text{get}(l,'title');
\LEG = \text{findobj}(l,'type','text');
\set(LEG,'FontSize',18)
\set(v,'string','Distributions','FontWeight','bold','FontSize',20);
\title('Collision pairs tester using 2^{20} samples','FontSize',24);
\grid on
\hold off

\% on figure(3): (histograms of the number of pairs collision)
\% using Sample size of 2^{20}.
\% The measured distribution are:
\% 1. RC4 keystream bytes at location (z1,z2)
\% 2. RC4 TKIP keystream bytes at location (z1,z2) when TSC0=0x00
\% and TSC1=0xFF

\textit{figure}(3)
\textit{fileID} = \text{fopen('collision\_rc4\_1\_2\_exp\_20.txt','r');}
\textit{rc4\_1\_2\_vec} = \text{fscanf(fileID,'%d\n');}
\text{fclose(fileID);}
\[m,x]=\text{hist}(rc4\_1\_2\_vec,12);
b1=\text{bar}(x,m/\text{trapz(x,m)},'b');
\hold on
\b = 8410000:200:8435000;
\mu = \text{mean}(rc4\_1\_2\_vec);
\sigma = \text{std}(rc4\_1\_2\_vec);
\f = \exp\left(-\frac{(b-\mu)^2}{2\sigma^2}\right)/\left(\sigma \sqrt{2\pi}\right);
p1=\text{plot}(b,f,'b','LineWidth',4);
\textit{fileID} = \text{fopen('collision\_tkip\_1\_2\_exp\_20.txt','r');}
\textit{tkip\_1\_2\_vec} = \text{fscanf(fileID,'%d\n');}
\text{fclose(fileID);}
\[m,x]=\text{hist}(tkip\_1\_2\_vec,12);
b2=\text{bar}(x,m/\text{trapz(x,m)},'y');
\b = 9720000:200:9840000;
\mu = \text{mean}(tkip\_1\_2\_vec);
\sigma = \text{std}(tkip\_1\_2\_vec);
\f = \exp\left(-\frac{(b-\mu)^2}{2\sigma^2}\right)/\left(\sigma \sqrt{2\pi}\right);
p2=\text{plot}(b,f,'Color',[1, 1, 0],'LineWidth',4);
\set(gca, 'FontSize', 20)
\xlabel('Number of colliding pairs','FontSize',20,'FontWeight','bold');
\ylabel('Normalized histogram','FontSize',20,'FontWeight','bold');
\axis([8200000 1000000 0 0.0002]);
\l=\text{legend}([p1,p2],'(z_1,z_2)','$TKIP (z_{32},z_{33})$');
v = \text{get}(l,'title');
\LEG = \text{findobj}(l,'type','text');
\set(LEG,'FontSize',18)
\set(v,'string','Distributions','FontWeight','bold','FontSize',20);
\title('Collision pairs tester using 2^{20} samples','FontSize',24);
\grid on

73
% on figure(4): (histograms of the number of pairs collision)
% using Sample size of 2^22.
% The measured distribution are:
% 1. RC4 keystream bytes at location (z_1,z_2)
% 2. RC4 keystream bytes at location (z_{100},z_{101})
% 3. RC4 TKIP keystream bytes at location (z_{32},z_{33}) when TSC0=0
% and TSC1=0x00

figure(4)

fileID = fopen('collision_tkip_32_33_exp_22.txt','r');
tkip_32_33_vec = fscanf(fileID,'%d
');
close(fileID);

[m,x]=hist(tkip_32_33_vec,12);
b1=bar(x,m/trapz(x,m),'r');

hold on
b = 134270000:1000:134360000;
mu = mean(tkip_32_33_vec);
sigma = std(tkip_32_33_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p1=plot(b,f,'r','LineWidth',4);

fileID = fopen('collision_rc4_1_2_exp_22.txt','r');
rc4_1_2_vec = fscanf(fileID,'%d
');
close(fileID);

[m,x]=hist(rc4_1_2_vec,12);
b2=bar(x,m/trapz(x,m),'b');

b = 134680000:1000:134820000;
mu = mean(rc4_1_2_vec);
sigma = std(rc4_1_2_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p2=plot(b,f,'b','LineWidth',4);

fileID = fopen('collision_rc4_100_101_exp_22.txt','r');
rc4_100_101_vec = fscanf(fileID,'%d
');
close(fileID);

[m,x]=hist(rc4_100_101_vec,12);
b3=bar(x,m/trapz(x,m),'g');

b = 134170000:1000:134260000;
mu = mean(rc4_100_101_vec);
sigma = std(rc4_100_101_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p3=plot(b,f,'Color', [0.3, 0.8, 0.3],'LineWidth',3);

set(gca, 'FontSize', 20)
xlabel('Number of colliding pairs','FontSize',20,'FontWeight','bold');
ylabel('Normalized histogram','FontSize',20,'FontWeight','bold');

title('Collision pairs tester using 2^22 samples','FontSize',24);
grid on
hold off

fileID = fopen('collision_tkip_1_2_exp_22.txt','r');
tkip_1_2_vec = fscanf(fileID,'%d
');
close(fileID);
% on figure(5): (histograms of the number of pairs collision)  
% using Sample size of $2^{28}$. 
% The measured distribution are: 
%  1. RC4 keystream bytes at location $(z_4,z_5)$  
%  2. RC4 keystream bytes at location $(z_{100},z_{101})$  
%  3. Random() on the range $[0,2^{16}-1]$

figure (5)

fileID = fopen('collision_test_random_28.txt','r');
random_vec = fscanf(fileID,'%lu
');
fclose(fileID);

[m,x]=hist(random_vec,12);
b1=bar(x,m/trapz(x,m),'FaceColor',[0, 0, 0]);
hold on
b = 549753000000:100000:549759000000;
mu = mean(random_vec);
sigma = std(random_vec);
f = exp(-(b-mu).^2./(2*sigmaˆ2))./(sigma*sqrt(2*pi));
p1=plot(b,f,'Color', [0, 0, 0],'LineWidth',4);

fileID = fopen('collision_rc4_100_101_exp_28.txt','r');
rc4_100_101_vec = fscanf(fileID,'%lu
');
fclose(fileID);

[m,x]=hist(rc4_100_101_vec,12);
b2=bar(x,m/trapz(x,m),'g');
mu = mean(rc4_100_101_vec);
sigma = std(rc4_100_101_vec);
f = exp(-(b-mu).^2./(2*sigmaˆ2))./(sigma*sqrt(2*pi));
p2=plot(b,f,'Color', [0, 0, 0],'LineWidth',4);

fileID = fopen('collision_rc4_4_5_exp_28.txt','r');
rc4_4_5_vec = fscanf(fileID,'%lu
');
fclose(fileID);

[m,x]=hist(rc4_4_5_vec,12);
b3=bar(x,m/trapz(x,m),'FaceColor',[0.6, 0, 0.8]);
mu = mean(rc4_4_5_vec);
sigma = std(rc4_4_5_vec);
f = exp(-(b-mu).^2./(2*sigmaˆ2))./(sigma*sqrt(2*pi));
p3=plot(b,f,'Color', [0.6, 0, 0.8],'LineWidth',3);

set(gca, 'FontSize', 20)
xlabel('Number of colliding pairs','FontSize',20,'FontWeight','bold');
ylabel('Normalized histogram','FontSize',20,'FontWeight','bold');
l=legend([p1,p2,p3],'Random', '(z_{100}),z_{101}','(z_{4},z_{5})');
v = get(l,'title');
LEG = findobj(l,'type','text');
set(LEG,'FontSize',18)
set(v,'string','Distributions','FontSize',20,'FontWeight','bold','FontSize',20);
title('Collision pairs tester using $2^{28}$ samples','FontSize',24);
grid on
hold off

% on figure(6): (histograms of the number of pairs collision)  
% using Sample size of $2^{32}$. 
% The measured distribution are: 
%  1. RC4 keystream bytes at location $(z_4,z_5)$

75
2. RC4 keystream bytes at location $(z_{100},z_{101})$
3. Random() on the range $[0,2^{16}-1]$

```matlab
figure (6)
fileID = fopen('collision_test_random_32.txt','r');
random_vec = fscanf(fileID,'%lu
');
fclose(fileID);
[m,x]=hist(random_vec,5);
b1=bar(x,m/sum(x),'FaceColor',[0, 0, 0]);
hold on
b = 140737430000000:1000000:1407375500000000;
mu = mean(random_vec);
sigma = std(random_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p1=plot(b,f,'Color', [0, 0, 0], 'LineWidth',4);

fileID = fopen('collision_rc4_100_101_exp_32.txt','r');
rc4_100_101_vec = fscanf(fileID,'%lu
');
fclose(fileID);
[m,x]=hist(rc4_100_101_vec,5);
b2=bar(x,m/sum(x),'FaceColor',[0, 0, 0], 'LineWidth',4);
b = 140737480000000:1000000:1407376000000000;
mu = mean(rc4_100_101_vec);
sigma = std(rc4_100_101_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p2=plot(b,f,'Color', [0, 0, 0], 'LineWidth',4);

fileID = fopen('collision_rc4_4_5_exp_32.txt','r');
rc4_4_5_vec = fscanf(fileID,'%lu
');
fclose(fileID);
[m,x]=hist(rc4_4_5_vec,5);
b3=bar(x,m/sum(x),'FaceColor',[0.6, 0,0.8], 'LineWidth',3);
b = 140737660000000:1000000:1407377700000000;
mu = mean(rc4_4_5_vec);
sigma = std(rc4_4_5_vec);
f = exp(-(b-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
p3=plot(b,f,'Color', [0.6, 0, 0.8], 'LineWidth',3);
set(gca, 'FontSize', 20)
xlabel('Number of colliding pairs','FontSize',20,'FontWeight','bold');
ylabel('Normalized histogram','FontSize',20,'FontWeight','bold');
l=legend([p1,p2,p3],'Random', '(z_{100},z_{101})', '(z_{4},z_{5})');
v = get(l,'title');
LEG = findobj(l,'type','text');
set(LEG,'FontSize',18)
set(v,'string','Distributions','FontWeight','bold','FontSize',20);
title('Collision pairs tester using 2^{32} samples','FontSize',24);
grid on
hold off
```
B.4 Fingerprint code

B.4.1 RC4_fingerprint.cpp

/* RC4_FP_test.cpp
This program runs 100 simulations.
In each simulation, it generates 2^("exp") samples of RC4 keystream bytes
at positions (z_first_location,z_first_location+i), and prints the "
Fingerprint"
of the sampled distribution, i.e. how many bins have exactly 1 sample in them
how many bins have exactly 2 samples in them and so on.
The results are saved to a file.
*/

#include <string.h>
#include <stdlib.h>
#include <random>
#include <iostream>

#define N (1 << 16) // 2^16
#define MAX_U64 0xffffffffffffffff // 2^64-1
#define NUM_OF_SIMULATIONS 100

/* Initializes random number generator */
std::random_device rd;
std::mt19937_64 mt(rd);
std::uniform_int_distribution<unsigned long long> dist(0, MAX_U64);

/* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which
in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i<=255; i++)
        S[i]=i;
    j=0;
    for (i=0; i<=255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
    }
}

/* Initialize the key with 16 bytes random values */
inline void create_key(void* key)
{
    int i;
    unsigned long long* key_ullp = (unsigned long long*) key;
    for (i=0; i < 2; i++)
        *(key_ullp+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name Exponent FirstByteLocation" where:
Name - The name of the execution file (rc4_fp)
Exponent - Log (on base 2) of the number of samples required.
Should be in the range [15,26]
FirstByteLocation - First byte location in the keystream where we
want to start counting the values for learning the distribution
at position (z_first_location,z_first_location+i)."
Should be in the range \([1,10,000]\).

```c
/*
 * CheckArguments(int argc, char* argv[], int& exp, int& first_byte_location)
 * 
 *     fprintf(stderr,"Bad call. Usage: Name Exponent FirstByteLocation.\n")
 *     return 1;
 * 
 * exp = atoi(argv[1]);
 * if (exp < 15 || exp > 26)
 *     fprintf(stderr,"Bad argument. Exponent should be in "
 *                   "the range \([15,26]\)\n")
 *     return 1;
 * 
 * first_byte_location = atoi(argv[2]);
 * if (first_byte_location < 1 || first_byte_location > 10000)
 *     fprintf(stderr,"Bad argument. FirstByteLocation should be in "
 *                   "the range \([1,10000]\)\n")
 *     return 1;
 * 
 * return 0;
 */

int main(int argc, char *argv[])
{
    unsigned int i, j, m;
    unsigned char S[256];
    unsigned char key_str[16];
    FILE *fp;

    unsigned short counter[N];
    unsigned short bins_counter[1000];
    unsigned short temp;
    char* temp_p = (char*)&temp;
    int exp, first_byte_location;
    char file_name[60];

    if (CheckArguments(argc,argv,exp,first_byte_location))
        return 1;

    sprintf(file_name,60,"rc4_fp_%d_%d_exp_%d.txt",first_byte_location,
            first_byte_location+1,exp);

    fp = fopen(file_name, "w");
    if (fp == NULL)
        {        fprintf(stderr, "Error opening file!\n");
            return (1);
        }

    // Running 100 simulations
    for (int iter=0; iter < NUM_OF_SIMULATIONS; iter++)
        {
            // Initializing the counters to zero
            memset(counter,0,sizeof(counter));
            memset(bins_counter,0,sizeof(bins_counter));

            // Generating 2^("exp") samples
            for (unsigned long long l=0;
                l < (unsigned long long) (1LL << exp); l++)
                {
                    create_key(key_str);
                    RC4_KSA(S,key_str);

                    // Generating the keystream bytes
                    for(i=0,j=0,m=0;m < (unsigned int) first_byte_location+1;m++)
                        { i = (i + 1) % 256;
                          j = (j + S[i]) % 256;
                          // swap s[i] and s[j]
                          Swap(S[i],S[j]);
                          
                          if (m== (unsigned int) first_byte_location-1)
```

78
{ temp = S[(S[i]+S[j]) % 256];
}
else if (m==*(unsigned int) first_byte_location)
{
    *(temp_p+1)=S[(S[i]+S[j]) % 256];
}

// "temp" is the value at (z_first_location,z_first_location+1)
// The byte at (z_first_location+1) chosen as the MSB.
counter[temp]++;

// Count bins with 0 items, then with 1 items, and so on...
for(m=0; m < N; m++)
{
    bins_counter[(counter[m])]++;
}

// Printing the results to a file
for (m=0; m < 1000; m++)
{
    fprintf(fp, "%u\t", bins_counter[m]);
}
 fprintf(fp, "\n");

fclose(fp);
return 0;
B.4.2 TKIP_fingerprint.cpp

#include <string.h>
#include <stdlib.h>
#include <random>
#include <iostream>

#define N (1 << 16) // 2^16
#define MAX_U32 0xffffffff // 2^32-1
#define NUM_OF_SIMULATIONS 100

/* Intializes random number generator */
std::random_device rd;
std::mt19937 mt(rd());

/* Swap between the values 'a' and 'b' */
inline void Swap(unsigned char& a ,unsigned char& b)
{
    unsigned char temp = a;
    a = b;
    b = temp;
}

/* Initialize the permutation in the array "S" using the key "key", which
in our case is 16 bytes long */
void RC4_KSA(unsigned char* S, const unsigned char* key)
{
    unsigned int i,j;
    for (i=0; i <255 ; i++)
        S[i]=i;
    j=0;
    for (i=0; i <255 ; i++)
    {
        j = (j + S[i]+ key[i % 16]) % 256;
        // swap s[i] and s[j]
        Swap(S[i],S[j]);
    }
}

/* Initialize the key.
The first 3 bytes of the key are always depend only on TSC0 and TSC1,
which in our case stayed constants.
The rest of the key (13 bytes) is assigned with random values */
inline void create_key(void* key)
{
    int i;
    unsigned int* key_uip = (unsigned int*) key;
    unsigned char* key_ucp = (unsigned char*) key;
    key_uip[3] = dist(mt) % 256;
    for (i=1; i < 4; i++)
        *(key_uip+i) = dist(mt);
}

/* Check that the arguments which were passed to this program are valid.
Properly call for this program should be in the following form:
"Name Exponent FirstByteLocation TSC0 TSC1" where:
Name - the name of the execution file (tkip_fp)
Exponent - Log (on base 2) of the number of samples required.
Should be in the range [15,26]
FirstByteLocation - First byte location in the keystream where we
want to start learning the distribution
*/
at position (first_location,first_location+1).
Should be in the range [1,10,000].

TSC0, TSC1 - The least-significant bytes (LSB) of the TKIP sequence counter "TSC".
Should be in the range [0,255].

*/
int CheckArguments(int argc, char* argv[],int & exp, int& first_byte_location,
int& tsc0,int& tsc1)
{
    {
        fprintf(stderr,"Bad call.\nUsage: Name Exponent FirstByteLocation TSC0 TSC1\\n" );
        return 1;
    }
    exp = atoi(argv[1]);
    if (exp < 15 || exp > 26)
    {
        fprintf(stderr,"Bad argument.\nExponent should be in "
        "the range [15,26]\n" );
        return 1;
    }
    first_byte_location = atoi(argv[2]);
    if (first_byte_location < 1 || first_byte_location > 10000)
    {
        fprintf(stderr,"Bad argument.\nFirstByteLocation should be in "
        "the range [1,10000]\n" );
        return 1;
    }
    tsc0 = atoi(argv[3]);
    tsc1 = atoi(argv[4]);
    if (tsc0 < 0 || tsc0 > 255 ||
    tsc1 < 0 || tsc1 > 255)
    {
        fprintf(stderr,"Bad argument.\nTSC value should be in "
        "the range [0,255]\n" );
        return 1;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    unsigned int i,j,m;
    unsigned char S[256];
    unsigned char key_str[16];
    FILE *fp;
    unsigned short counter[N];
    unsigned short bins_counter[1000];
    unsigned short temp;
    char* temp_p = (char*)&temp;
    int exp, first_byte_location,tsc0,tsc1;
    char file_name[60];

    if (CheckArguments(argc,argv,exp,first_byte_location,tsc0,tsc1))
        return 1;
    // Initializing the first 3 bytes of the key, using TSC0 and TSC1
    key_str[0] = (unsigned char) tsc1;
    key_str[1] = ((unsigned char)tsc1|0x20)&0x7f;
    key_str[2] = (unsigned char) tsc0;
    snprintf(file_name,60,"tkip_fp_%d_%d_exp_%d.txt",first_byte_location,
    first_byte_location+1,exp);
    fp = fopen(file_name, "w");
    if (fp == NULL)
    {
        fprintf(stderr, "Error opening file!\n" );
        return (1);
    }
    // Running 100 simulations

for (int iter=0; iter < NUM_OF_SIMULATIONS; iter++)
{
    // Initializing the counters to zero
    memset(counter,0,sizeof(counter));
    memset(bins_counter,0,sizeof(bins_counter));

    // Generating \(2^\text{"exp"}\) samples
    for (unsigned long long l=0; l < (unsigned long long)(1LL << exp); l++)
    {
        create_key(key_str);
        RC4_KSA(S,key_str);

        // Generating the keystream bytes
        for(i=0,j=0,m=0;m < (unsigned int) first_byte_location+1;m++)
        {
            i = (i + 1) % 256;
            j = (j + S[i]) % 256;
            // swap s[i] and s[j]
            Swap(S[i],S[j]);

            if (m== (unsigned int) first_byte_location-1)
                temp = S[(S[i]+S[j]) % 256];
            else if (m==(unsigned int) first_byte_location)
                *(temp_p+1)=S[(S[i]+S[j]) % 256];
        }

        // "temp" is the value at \((z\text{\_first\_location},z\text{\_first\_location}+1)\)
        // The byte at \((z\text{\_first\_location}+1)\) chosen as the MSB.
        counter[temp]++;
    }

    // Count bins with 0 items, then with 1 items, and so on...
    for(m=0;m < N; m++)
    {
        bins_counter[(counter[m])]++;
    }

    // Printing the results to a file
    for (m=0; m < 1000; m++)
    {
        fprintf(fp,"%u\t",bins_counter[m]);
    }
    fprintf(fp,"\n");
}

fclose(fp);
return 0;
B.4.3 FingerprintTester.m (Matlab script)

```matlab
clear all
clc
format long

% This script reads the "FingerPrint" data from the output files,
% and displays the results on a graph.
% Each file contains the "FingerPrint" of one empirical distribution,
% when using 100 simulations with S=2^21 samples each.
% The Files are:
% 1. "tkip_fp_1_2_exp_21.txt" - RC4 TKIP keystream bytes at location (z1,z2)
%   when TSC0=0x00 and TSC1=0xFF
% 2. "rc4_fp_1_2_exp_21.txt" - RC4 keystream bytes at location (z1,z2)
% 3. "tkip_fp_32_33_exp_21.txt" - RC4 TKIP keystream bytes at location (z32,z33)
%   when TSC0=0x00 and TSC1=0x00
% 4. "rc4_fp_100_101_exp_21.txt" - RC4 keystream bytes at location (z100,z101)
%
size = [1000 100];
NUM_OF_SAMPLES= 2^21;
NUM_OF_BINS = 2^16;
prob_bin = 1/NUM_OF_BINS;

fileID = fopen('tkip_fp_1_2_exp_21.txt','r');
A = fscanf(fileID,'%d',size);
fclose(fileID);

fileID = fopen('rc4_fp_1_2_exp_21.txt','r');
B = fscanf(fileID,'%d',size);
fclose(fileID);

fileID = fopen('tkip_fp_32_33_exp_21.txt','r');
C = fscanf(fileID,'%d',size);
fclose(fileID);

fileID = fopen('rc4_fp_100_101_exp_21.txt','r');
D = fscanf(fileID,'%d',size);
fclose(fileID);

A_t=A';
A_m=round(mean(A_t));
A_std=round(std(A_t));

B_t=B';
B_m=round(mean(B_t));
B_std=round(std(B_t));

C_t=C';
C_m=round(mean(C_t));
C_std=round(std(C_t));

D_t=D';
D_m=round(mean(D_t));
D_std=round(std(D_t));

bar_matrix=zeros(1000,5);
bar_matrix(:,1)=A_m';
bar_matrix(:,2)=B_m';
bar_matrix(:,3)=C_m';
bar_matrix(:,4)=D_m';
unif_bar=zeros(1000,1);
for i=0:999
    unif_bar(i+1)=nchoosek(NUM_OF_SAMPLES,i)*(prob_bin).^(i)*(1-prob_bin).^(NUM_OF_SAMPLES-(i))*NUM_OF_BINS;
end
```

83
bar_matrix(:,5)=unif_bar;
x=0:1:999;
b=bar(x,bar_matrix);
set(b(1),’FaceColor’,’y’,’EdgeColor’,[1,0.8,0])
set(b(2),’FaceColor’,’b’,’EdgeColor’,’b’)
set(b(3),’FaceColor’,’r’,’EdgeColor’,’r’)
set(b(4),’FaceColor’,[0,0.8,0],’EdgeColor’,[0,0.8,0])
set(b(5),’FaceColor’,[0 0 0],’EdgeColor’,[0,0,0])
set(gca, ‘FontSize’, 20)
grid on

title(’Fingerprint, using N=2^{16} (number of domain elements) and S=2^{21} (sample size)’,’FontSize’,22,’FontWeight’,’bold’)
xlabel(’$i$ (The entry of the fingerprint vector)’,’Interpreter’,’LaTex’,’FontSize’,20,’Rotation’,0);
ylabel(’$\overline{F_{\hat{D}}}(i)$’,’Interpreter’,’LaTex’,’FontSize’,20,’Rotation’,0);
axis([9 280 0 5000]);
set(y, ’Units’, ’Normalized’, ’Position’, [0.1, 0.5, 0]);

l=legend([b(1),b(2),b(3),b(4),b(5)],’(TK_1,TK_2)’,’(Z_1,Z_2)’,’(TK_{32},TK_{33})’,’(Z_{100},Z_{101})’,’Uniform distribution (expectation value)’);
LEG = findobj(l,’type’,’text’);
set(LEG,’FontSize’,16)
v = get(l,’title’);
set(v,’string’,’Distributions’,’FontWeight’,’bold’,’FontSize’,20);