Sheet 2

Dated 16.11.2021

Compiled - Deo, Pranav | Nurt, Aurika

Exercise 1: Postprocessing of True Random Number Generators - Theoretical Background

1.1 In which cases is post-processing required for true random number generators?

TRNG usually use some natural physical phenomenon in order to extract the entropy and be the source of true randomness [such as rae conditions, air turbulence in HDDs, mouse movement or any other noise], that is to make the system non-deterministic.

- Nevertheless, however random may the the natural phenomenon source may seem, the digitalization of a physical phenomenon may introduce some form of bias, usually some statistical bias.
- There might be implementation errors with the ICs or the ASICs (application specific ICs) that are used to achieve the digitalization of the physical phenomenon, or the components could be defective (i.e parasitic non ideal components).
- Sometimes sampling can introduce correlation.
- Thus, in order to resolve these issues, and covert any biased randomness (not really random then is it ?xD) bits to unbiased randomness, we require post processing techniques.

1.2 Name three well known post processing techniques? What are their potential shortcomings?

A. Von-Neumann Correction Method

- Take two random bits and compare, if they are same discard the bits, and if not then use the first bit.
- Why? Suppose that the `Pb(0)==x && Pb(1)==y` such that `x!=y`, then the `Pb(01)==Pb(10)==x*y`. The `Pb(00)==x^2 || Pb(11)==y^2` are discarded.

B. Parity Based Post-Processing

- Break input stream into chunks each of a `n` bits.

- Compute a parity bit for each chunk by performing an XOR operation of all the bits in stream, discard the chunk and then use the parity bit.
- Why? Because for every `n` bit chunk taken, there happen to be exactly 2^(n-1) zero values and 2^(n-1) one values. As such performing the XOR operation increases chaos (entropy) leading to randomness.
- Problems - reduces the stream to 1/n, requires heavy computation for the same.

C. Universal hashing

- Use hashing techniques (SHA 1, MD5) to convert arbitrary length random bits into a fixed length stream.
- Problem - Computationally heavy, more the bits, more heavy.

Exercise 2

TRNG 1

```
00100000000011111100000000100100
01111111111111100000100011010101
10000001110111111111111111100100
00000011111111001011101111111111
11111011111011110111010111101111
```

TRNG 2

```
00110110100110110010000111000010
00111110001101111101111000100001
01001000110100111110000111001100
00111000000001100010110001101000
11010110101010010010101010111001
```

2.1 Compare quality of given TRNGs, list observations.

https://medium.com/unitychain/provable-randomness-how-to-test-rngs-55ac6726c5a3

Note:: How do we even really comprehend if something is random?

is `11111` random or is `0101010010101000101` random? We cant say really by observation only!

Popular tests to check randomness:

- NIST RNG

- Dieharder test suite
- Knuth test

To test randomness of a sequence, we first start by analyzing a the source of entropy, and then we go after the 'deterministic' algorithm that uses the entropy seed and expands it into a sequence of keys.

Using Block Frequency Testings::

- Considering the output of TRNG 1 as a block of streams, we will find the probability of `1` and `0` in each sub-block. We will then evaluate the mean ratios and see what the probability of each bit is.

- We are looking for the Pb(1) and Pb(0) be as close to 0.5 as much as possible. When both the probabilities are 0.5, we can say that there is true randomness as the chance of both occurrences are equal.

- Should the probability skew, then we can say that this one

Considering the output of TRNG 1 block of stream

```
[Sub-block 1]   00100000000001111110000000000100100 ---> Pb(1)=  9/32 = 0.281 && Pb(0) == 23/32 = 0.718

[Sub-block 2]   01111111111111100000100011010101   ---> Pb(1)=  20/32 = 0.625 && Pb(0) == 12/32 = 0.375

[Sub-block 3]   10000001110111111111111111100100   ---> Pb(1)=  21/32 = 0.656 && Pb(0) == 11/32 = 0.343

[Sub-block 4]   00000011111111001011101111111111   ---> Pb(1)=  22/32 = 0.687 && Pb(0) == 10/32 = 0.312

[Sub-block 5]   11111011111011110111010111101111   ---> Pb(1)=  26/32 = 0.812 && Pb(0) == 6/32 = 0.187

Thus the total average Pb(1) = .6122
Thus the total average Pb(0) = .3878
```

Considering the output of TRNG 2 block of stream

```
[Sub-block 1]   00110110100110110010000111000010 ---> Pb(1)=  14/32 = 0.4375 && Pb(0) == 18/32 = 0.5625

[Sub-block 2]   00111110001101111101111000100001 ---> Pb(1)=  14/32 = 0.4375
```

```
&& Pb(0) == 18/32 = 0.5625

[Sub-block 3]   01001000110100111110000111001100 ---> Pb(1)=  17/32 = 0.53125
&& Pb(0) == 15/32 = 0.46875

[Sub-block 4]   00111000000001100010110001101000 ---> Pb(1)=  10/32 = 0.3125
&& Pb(0) == 22/32 = 0.6875

[Sub-block 5]   11010110101010010010101010111001 ---> Pb(1)=  17/32 = 0.53125
&& Pb(0) == 15/32 = 0.46875

Thus the total average Pb(1) = .45
Thus the total average Pb(0) = .55
```

Observations:

- Sequence of TRNG 2 seems to be more random than TRNG 1 since the Pb(1) and Pb(0) are more close to being 0.5, thus there is less disparity in the frequency occurrence of 0 and 1.
- Even at a quick glance at the sequence, the transitions of `1 -> 0` or `0 -> 1` (aka 'runs'), are less for sequence of TRNG 1. Less number of runs indicate a poor randomness.
- Less number of runs in TRNG 1 also indicate bigger longest-run-of-m bits, i.e. the repletion of big blocks of either 1 or 0 are more in TRNG1 sequence.
  (here we dont know if this large blocks of a certain bit are truly random or not since we dont really know the source os entropy that resulted in the sequence).
- *TRNG 2 sequence appears to be more random and hence of better quality than TRNG 1 sequence.*

2.2 Apply the Von-Neumann correction to the random sequences

2.3 Apply parity based method on the sequence given in the task using a chunk size of 16

Method

- Divide stream in n bits.
- Calculate XOR calculation of the bits, and discard the chunk.

```
XOR TABLE REF

A   B  A_XOR_B
```

```
0   0     0
0   1     1
1   0     1
1   1     0
```

Dividing the Sequence of TRNG 1 into chunk size of 16 ::

```
n=16

0010000000001111      1100000000100100   ---->    1     0
   (Chunk 1)               (Chunk 2)

0111111111111110      0000100011010101   ---->    0     0


1000000111011111      1111111111100100   ---->    1     0


0000001111111100      1011101111111111   ---->    0     0


1111101111101111      0111010111101111   ---->    0     0


Thus the sequence is reduced to :

00100000000011111100000000100100          10
01111111111111100000100011010101          00
10000001110111111111111111100100   -->    01
00000011111111001011101111111111          00
11111011111011110111010111101111          00
```

Dividing the Sequence of TRNG 2 into chunk size of 16 ::

```
0011011010011011      0010000111000010   ---->    1     1


0011111000110111      1101111000100001   ---->    0     0

```

```
0100100011010011      1110000111001100   ---->    1     0


0011100000000110      0010110001101000   ---->    1     0


1101011010101001      0010101010111001   ---->    1     0


Thus the sequence is reduced to :


00110110100110110010000111000010          11
00111110001101111101111000100001          00
01001000110100111110000111001100   -->    10
00111000000001100010110001101000          10
11010110101010010010101010111001          10
```

Task 3: Cellular Automata Shift Register

(175)base_10 = (10101111)base_2

(242)base_10 = (11110010)base_2

Table 1: Cellular Automata Shift Register

Rule List	**175**	**242**	**175**	**175**	**242**	**175**	**242**	**242**
State 0	0	1	1	1	0	1	1	0
State 1	1	0	1	1	1	1	1	1

Rule Table for (175)base_10

Number	**Neighborhood**	**Rule Set**
7	1 11	1
6	1 10	0
5	1 01	1
4	1 00	0
3	0 11	1
2	0 10	1
1	0 01	1

Number	Neighborhood	Rule Set
0	0 00	1

Rule Table for (242)base_10

Number	Neighborhood	Rule Set
7	1 11	1
6	1 10	1
5	1 01	1
4	1 00	1
3	0 11	0
2	0 10	0
1	0 01	1
0	0 00	0

Task 4 : PUFs : Physically Unclonable Functions

4.1 Three PUF application and how PUFs are used for it?

PUFs can be used for

- Identification and Authentication
- Storing keys and hashes
- Random Number Generators

4.2 Explain How optical PUFs work?

Optical PUFs work by scattering light over an transparent material that has some randomly scattered opaque spots/ particles scattered all over.

The opaque particle essentially block the light waves, and the result is a unique pattern (aka speckle pattern) that can be obtained on the other side of the transparent material.

This pattern is can be recorded to form a response database, and can be post processed as well.

Optical PUFs have quite a few issues:

- They require the availability of optical devices and optical readers.

- The optical measurements taken should be quite precise in order to create a proper response challenge database and later identify the challenge.

4.3 Think of a way of creating an optical PUF yourself with simple means?

4.4 Explain how the ring oscillator PUF works? What it it's problem?