Hardware_Security_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

```
001000000000111111000000000100100
011111111111111000001000011010101
100000011101111111111111111100100
000000111111110010111011111111111
111110111110111101110101111101111
```

TRNG 2

```
001101101001101100100000111000010
001111100011011111101111000100001
010010001101001111100001111001100
001110000000011000101100011010100
110101101010100100101010101111001
```

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]   00100000000011111100000000100100 ---> 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]   0011100000000110001011000110100 ---> 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

- The von Neumann correction is used to remove any bias from a pseudo-random bit stream.

- It takes two input bits and outputs a single bit, only when there is a transition from the first bit to second.

- That is if the two input bits are same, then they are discarded and if they are not, then first bit is selected as output.

- Ideally the two bits are selected randomly, but for the sake of simplicity

TRNG 1

```
input   00 10 00 00 00 00 11 11 11 00 00 00 00 10 01 00
output     1                                 1  0
```

```
input    01 11 11 11 11 11 11 10 00 00 10 00 11 01 01 01
output    1                       1        1         0  0  0


input    10 00 00 01 11 01 11 11 11 11 11 11 11 10 01 00
output    1        0     0                          1  0


input    00 00 00 11 11 11 11 00 10 11 10 11 11 11 11 11
output                            1     1


input    11 11 10 11 11 10 11 11 01 11 01 01 11 10 11 11
output          1        1           0     0  0     1
```

TRNG 2

```
input    00 11 01 10 10 01 10 11 00 10 00 01 11 00 00 10
output          0  1  1  0  1        1        0           1


input    00 11 11 10 00 11 01 11 11 01 11 10 00 10 00 01
output             1        0           0     1     1     0


input    01 00 10 00 11 01 00 11 11 10 00 01 11 00 11 00
output    0     1           0              1     0


input    00 11 10 00 00 00 01 10 00 10 11 00 01 10 10 00
output          1              0  1        1        0  1  1


input    11 01 01 10 10 10 10 01 00 10 10 10 10 11 10 01
output     0  0  1  1  1  1  0        1  1  1  1     1  0
```

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 :

0010000000001111100000000100100              10
0111111111111110000010001101010101          00
1000000111011111111111111111100100    -->    01
0000001111111100101110111111111111          00
1111101111101111011101011110111111          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 :


0011011010011011001000011100010          11
0011111000110111110111100010001          00
0100100011010011111000011100110    -->   10
0011100000000110000101100011010000       10
1101011010101001001010101010111001       10
```

2.4 What do you observe when comparing these two post-processing methods?

- In the von Neumann correction, the TRNG1 sequence was reduced by 0.1375 [22/160] and TRNG sequence was reduced by 0.2437 [39/160].

- This sheds light on our earlier observation that TRNG 1 contains more sequence of constant m-bit blocks such as '111' or '0000'.

- Thus the output of von Neumann method depends on the quality of the input sequence, and while truly random sequences will be reduced by 0.25, others will be greatly reduced.

Note: why does this happen?
[https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_520]

- In the parity based correction method, teh sequence was always reduced by 1/n, i.e 1/16 in our example.

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
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?

https://www.sciencedirect.com/topics/computer-science/ring-oscillator
https://www.researchgate.net/publication/355078122_Enhancing_the_Performance_of
_Lightweight_Configurable_PUF_for_Robust_IoT_Hardware-Assisted_Security

- A ring oscillator is a device that consists of a odd number of NOT gates laid in a ring fashion. The output of a ring oscillator varies between two voltage levels, each representing a binary 0 or 1 (true or false).

- The reason that we have multiple numbers of NOT gate, and exactly odd number of NOT gates, is that an odd number of odd gates means that the output is exactly the same as having one NOT gate, but the additional gates add a certain amount of delay.

- this delay is random and cannot be controlled in any way (thus a good source of random entropy source).
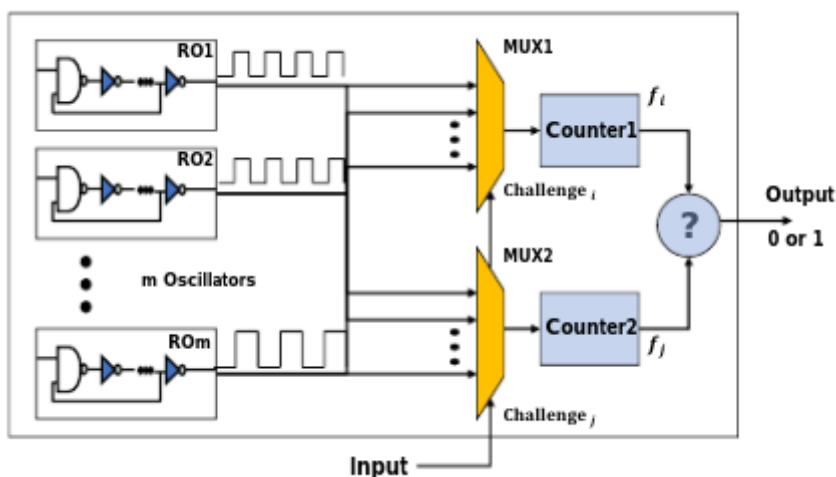
- A ROPUF (Ring oscillator puf) relies on mapping **m** challenge bits to **n** response bits.

- The response bits **r$_i$** (see image above) are usually determined by the manufacturer during the manufacturing process, as the fabrication method used usually leads to slight changes in the frequency of each ring oscillator.

- We have two challenge bits, **challenge$_i$** and **challenge$_j$** , which are fed into the multiplexer.

- These challenge bits decide which ring oscillator will be selected by the multiplexers.

- The relative frequencies of the selected ring oscillators, say **f$_i$** and **f$_j$**, are then compared, by comparing the relative number of clock cycles over a given time span.

- The response bit is **r$_i$** generated as follows:

```
ri = {1, if fi >= fj || 0, otherwise}
```

Problems with ROPUFS

- The problems with ROPUFs may lie in the ring oscillators itself.

- By design the ROs might not all work, and only some ROs might generate any response. This leads to a certain level dependency.

- Jitter is a common issue with ROs, such that deviation in temperatures can cause jitter, which may change the way an RO reacts to a challenge bit or change the frequency of RO causing undesired outputs.