University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Cybersecurity

Temperature attacks on TRNG and QRNG devices

Supervisor
Prof. Mauro Conti
University of Padova

Co-supervisor
Prof. Julio Hernandez Castro
University of Kent

Master Candidate
Ludovica Barsi

Student ID
2004063

Academic Year
2022-2023

“”

Abstract

The generation of completely random sequences is a crucial point for the security of systems and technological infrastructures: in fact, we need purely random sequences for example for the generation of cryptographic keys , values intended to be used only once (nonces) or initial values (IV) necessary to check the integrity of the message received. These sequences can be generated either by software or by hardware. For the latter, the generation of random numbers is possible thanks to a physical (or quantum) process that let the produced sequences to be completely random and therefore to be not predictable at all. Indeed, the big problem related to the generation of random numbers is that if an attacker were able to predict what the sequence produced in output will be, he could be able to compromise the entire system.

In this work, carried out in collaboration with the University of Kent (Kent, UK), we tried to perform two different temperature attacks on two different hardware devices: in particular, we worked on a True Random Number Generator (TRNG) named OneRNG and a Quantum Random Number Generator (QRNG) named ComScire model PQ32MU. For the first attack, we brought both devices to very low temperatures (reaching a maximum of -30°C) while for the second attack, both devices were brought to a maximum temperature of 80°C and 170°C respectively.

The goal of these attacks was to measure not only which were the lowest and highest critical temperatures at which devices would stop generating truly random data, but also to measure which of the two devices was more resilient to thermal change. A thorough analysis of the results obtained follows, in which we will analyze each file produced by the two devices. For this scope, five different statistical tests were used: Ent, Booltest, PractRand, TestU01 and BitRepsv1.0, an innovative statistical test created by the University of Kent which, using the Bloom Filter, is able to provide a very accurate analysis of the results.

Contents

Abstract v

List of figures ix

List of tables xiii

Listing of acronyms xv

1 Introduction 1

2 Related Work 3

3 Main Concepts 5
- 3.1 Formal Definition of Random Numbers . . . 5
  - 3.1.1 The Continuous Uniform Distribution . . . 5
- 3.2 Random Number Generators . . . 6
  - 3.2.1 Main features . . . 7
  - 3.2.2 Different Random Number Generators in commerce . . . 7

4 The OneRNG TRNG Device 9
- 4.1 Device Design . . . 9
  - 4.1.1 Avalanche Diode . . . 9
  - 4.1.2 Channel-Hoppin Radio Receiver . . . 11
  - 4.1.3 Data Whitening . . . 11
- 4.2 Device Usage . . . 12
- 4.3 Data Collection . . . 12
  - 4.3.1 Testing Environment . . . 12
  - 4.3.2 Software Installation . . . 13
  - 4.3.3 Device Operation . . . 13
  - 4.3.4 Collection Process . . . 13

5 The ComScire PQ32MU QRNG Device 17
- 5.1 Device Design . . . 17
  - 5.1.1 Ring Oscillator . . . 18
  - 5.1.2 The Quantum-Shot Noise . . . 19
- 5.2 Device Usage . . . 20

5.3 Data Collection 21

6 Statistical Tests 25
6.1 Ent 25
6.2 BoolTest 27
6.3 TestU01 29
6.4 PractRand 31
6.5 BitReps 33
6.5.1 Mathematical Theory 33
6.5.2 The Bloom Filter 34
6.5.3 Main Features 36
6.5.4 Result interpretation 39

7 Temperature Attacks 41
7.1 The cooling attack 41
7.1.1 Setup 41
7.1.2 The attack 42
7.2 The heating attack 44
7.2.1 Setup 44
7.2.2 The attack 46

8 Results Analysis 49
8.1 Results obtained with devices in normal conditions 50
8.2 Results obtained with devices under attack 53
8.2.1 The cooling attack results 54
8.2.2 The heating attack results 55
8.3 Result Analysis 57
8.4 BitReps 60

9 Conclusion 79

References 81

Acknowledgments 83

Listing of figures

3.1 Probability Density Function of the Uniform Distribution 6

4.1 OneRNG Device 10
4.2 Avalanche diode in reverse bias conditions 10

5.1 PQ32MU QRNG ComScire Device 18
5.2 Ring Oscillator Diagram 19

6.1 Ent statistical test output 27
6.2 BoolTest statistical test output 28
6.3 TestU01 Rabbit statistical test output 30
6.4 PractRand statistical test output 32
6.5 BitReps Calculator Window 37
6.6 BitReps Analyzer Window 38
6.7 BitReps statistical test output 39

7.1 Setup of Cooling experiment 42
7.2 ComScire device after Cooling experiment 43
7.3 OneRNG Device after Cooling experiment 44
7.4 Setup of Heating experiment 45
7.5 Heating Attack on TRNG OneRNG device 47
7.6 Heating Attack on QRNG ComScire device 48

8.1 OneRNG and Comscire data files analyzed with Ent Statistical Test 51
8.2 ComScire data file analyzed with BoolTest Statistical Test 52
8.3 OneRNG cmd0 data file analyzed with BoolTest Statistical Test 52
8.4 OneRNG cmd1 data file analyzed with BoolTest Statistical Test 53
8.5 OneRNG cmd3 data file analyzed with BoolTest Statistical Test 53
8.6 OneRNG cmd6 data file analyzed with BoolTest Statistical Test 54
8.7 OneRNG cmd7 data file analyzed with BoolTest Statistical Test 54
8.8 ComScire data file analyzed with TestU01 Rabbit Battery Statistical Test . . . 55
8.9 OneRNG cmd0 data file analyzed with TestU01 Rabbit Battery Statistical Test 56
8.10 OneRNG cmd6 data file analyzed with TestU01 Rabbit Battery Statistical Test 57
8.11 OneRNG cmd7 data file analyzed with TestU01 Rabbit Battery Statistical Test 57
8.12 ComScire data file analyzed with PractRand Statistical Test 58
8.13 OneRNG cmd0 data file analyzed with PractRand Statistical Test 59

8.14 OneRNG cmd1 data file analyzed with PractRand Statistical Test 60
8.15 OneRNG cmd3 data file analyzed with PractRand Statistical Test 61
8.16 OneRNG cmd6 data file analyzed with PractRand Statistical Test 64
8.17 OneRNG cmd7 data file analyzed with PractRand Statistical Test 65
8.18 ComScire data file analyzed with Ent Statistical Test when the device is under the cooling attack 65
8.19 ComScire data file analyzed with BoolTest Statistical Test when the device is under the cooling attack 66
8.20 ComScire data file analyzed with Rabbit TestU01 Statistical Test when the device is under the cooling attack 66
8.21 ComScire data file analyzed with PractRand Statistical Test when the device is under the cooling attack 67
8.22 OneRNG data file analyzed with Ent Statistical Test when the device is under the cooling attack 67
8.23 OneRNG cmd0 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack 68
8.24 OneRNG cmd6 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack 68
8.25 OneRNG cmd7 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack 69
8.26 OneRNG cmd6 and cmd7 data file analyzed with TestU01 Rabbit Battery Statistical Test when the device is under the cooling attack 69
8.27 OneRNG cmd0 data file analyzed with PractRand Battery Statistical Test when the device is under the cooling attack 70
8.28 OneRNG cmd6 and cmd7 data files analyzed with PractRand Statistical Test when the device is under the cooling attack 71
8.29 ComScire data files analyzed with Ent Statistical Test when the device is under the heating attack 72
8.30 ComScire data files analyzed with BoolTest Statistical Test when the device is under the heating attack (temperature=180°C) 72
8.31 ComScire data files analyzed with BoolTest Statistical Test when the device is under the heating attack (temperature=200°C) 72
8.32 ComScire data files analyzed with PractRand Statistical Test when the device is under the heating attack 73
8.33 ComScire data files analyzed with TestU01 Rabbit Battery Statistical Test when the device is under the heating attack 74
8.34 OneRNG data files analyzed with Ent Statistical Test when the device is under the heating attack 74
8.35 OneRNG cmd0 data files analyzed with BoolTest Statistical Test when the device is under the heating attack 75

8.36 OneRNG cmd6 data files analyzed with BoolTest Statistical Test when the device is under the heating attack 75
8.37 OneRNG cmd7 data files analyzed with BoolTest Statistical Test when the device is under the heating attack 75
8.38 OneRNG cmd0 data files analyzed with PractRand Statistical Test when the device is under the heating attack 76
8.39 OneRNG cmd6 data files analyzed with PractRand Statistical Test when the device is under the heating attack 76
8.40 OneRNG cmd7 data files analyzed with PractRand Statistical Test when the device is under the heating attack 77

Listing of tables

4.1 OneRNG modes of operation. Source: www.moonbaseotago.com/onerng/theory 12

8.1 Bitreps, block size=32 bits, error rate= 0.00001, expected distribution [79480, 146] , expected duplicates 79837, Expected false positives 20 62
8.2 Bitreps, block size=64 bits, error rate= 0.00001, expected distribution [8] , expected duplicates 0, Expected false positives 10 62
8.3 Bitreps, block size=128 bits, error rate= 0.00001, expected distribution [5] , expected duplicates 0, Expected false positives 5 62
8.4 Bitreps, block size=256 bits, error rate= 0.00001, expected distribution [-] , expected duplicates 0, Expected false positives 3 63
8.5 Bitreps, block size=512 bits, error rate= 0.00001, expected distribution [5] , expected duplicates 0, Expected false positives 1 63

Listing of acronyms

RNG .... Random Number Generator

PDF .... Probablity Density Function

IC .... Integrated Circuit

PRNG .... Pseudo-Random Number Generators

TRNG .... True Random Number Generators

QRNG .... Quantum Random Number Generators

IV .... Initial Values

BS .... Block Size

1
Introduction

Random numbers are nowadays an essential resource given their use in the most diverse applications: indeed, random numbers are needed in statistical analysis [1], probability theory [2] as well as in modern-day computer simulations [3]. In particular, they are at the basis of today's digital encryption: acting as 'nonces', cryptographic keys, or initial values they are at the core of every cryptographic algorithm. The generation of these numbers is carried out either by specific software (in this case we are referring to pseudo-random numbers) or by hardware which exploit the physical (or quantum) properties of some components which by their nature are random, such as the avalanche diode or quantum shot noise.

It is important to underline that although there exist many different devices for generating random numbers, not all of them are reliable in the same way: that is to say, it is also essential to pay attention to the quality of the numbers that are produced, as not always a number that at first sight seems random it really is. There are a myriad of reasons why it could happen that a device does not behave as it should: this could be due to a malfunction of one of its components, to an algorithm not implemented correctly as well as to an attack carried out by an external attacker. To overcome this problem, statistical tests have been created capable of monitoring the quality of these numbers: among these we mention the most famous ones such as Ent, Dieharder or TestU01.

The aim of this thesis is to analyze the behavior of different hardware devices, specifically, a True Random Number Generator (TRNG) named 'OneRNG' and a Quantum Random Number Generator (QRNG) named 'ComScire PQ32MU', while these devices are under at-

tack. We performed two different temperature attacks: the first one aims to bring both devices to very low temperatures while the second one aims to bring them to an extremely high temperature up to when they stop their functioning. In both cases our objective is to measure how the devices react to such attacks and if and how much the entropy they are generating is influenced from these attacks. This analysis can be done by monitoring the quality of the random numbers generated by these devices through different statistical tests: for this work, we chose four of the most common statistical tests in commerce: Ent, BoolTest, PractRand and TestU01.

Moreover, a deep analysis of the statistical test BitReps is done: a novel statistical test developed by the University of Kent which exploits the Bloom Filter in order to check if there are any repetitions at bit level.

Chapter 2 presents the related work to this thesis; in Chapter 3 it is included a brief explanation of the main concepts needed in order to understand what a random number is and which are the main random number generators in commerce; Chapters 4 and 5 explain in detail the two devices we are going to use, which are their source of entropy and how we can use them in order to generate and collect data. In Chapter 6 are presented the main statistical tests with which we are going to perform the analysis of the results while Chapter 7 shows the two attacks we have performed. Chapter 8 shows the results and makes a comparison obtained by analysing the data before and after performing the attacks with the several statistical tests mentioned above while Chapter while chapter 9 is left for drawing conclusions.

2
Related Work

As the number of RNG hardware devices grew, so to did the number of attempted attacks. This is also true with regard to the creation of new statistical tests, as some peculiar attacks are so well performing that it is increasingly difficult to identify them. The aim of these tests is to try to recognize sequences related to each other (and therefore under attack) in the shortest possible time, in order to avoid that the compromised sequence can affect the application for which it is used.

In a 2016 article written by B. Yang [4], an efficient method to detect attacks using on-the-fly tests discovered which could be directly implemented in the RNG hardware. In particular, almost all low-cost attacks (such as changing the temperature, voltage or clock frequency) were attempted on the RNG device, subsequently several statistical tests were processed on input data with the aim of selecting the most accurate features (in terms of lowest percentage of error) for detecting attacks. As a result, it was demonstrated that on-the-fly tests are capable of detecting an attack as soon as a specific pattern of bits is identified in the sequence. This would allow to immediately stop the entropy source and avoid that the compromised sequences can affect the security of the device.

Another interesting result was found in [5], where V. Govindan et al. pointed out the inability of some statistical TRNG tests (among which we mention the NIST and Dieharder tests) to recognize if a particular TRNG device is under attack or not. To test this, they developed a "Hardware Trojan Horse" (HTH) based attack on the TRNG of a Field Programmable Gate Array (FPGA) cryptosystem, on which they were able to introduce a 96-bit fixed pattern at

the beginning of 100 consecutive 2^{18} bit-sized bitstream chunks. The HTH is activated when an internal temperature of 42°C is reached, and once the attack has been performed, the HTH returns to its inactive state until it is triggered again. It has been shown, accordingly to the birthday paradox, that in the presence of active HTH the probability of colliding bit sequences is of 100%, but no one of the statistical tests that have been performed on the device was able to point out that an attack was in progress.

Researchers in [6] exhibit a parametric hardware Trojan for a Ring Oscillator (RO-based) TRNG presented in [7]. They demonstrate that at high temperature the entropy source is disable to trigger the Trojan, and so it produces non-random and predictable results, but operates properly under normal environmental conditions. They achieved the rapid collapse behavior in a few cycles on the RO construction at high temperature, which is not useful to generate random bits to provide enough entropy.

Authors in [8] investigate the experimental evaluation of a TRNG based on the Transition Effect Ring Oscillator (TERO), which is originally proposed in [9]. They focused on the robustness of TERO TRNG against low temperature attacks and underpower attack. In this vein, they demonstrated that in their TERO TRNG difference between low-temperature attack and normal operation was fairly low, but it was vulnerable to underpower attack. Authors also clarified that lower voltage are known to slow down the circuit, which is influenced on the decrease of the number of oscillations and this consequently decrease of randomness.

3
Main Concepts

In this chapter the main concepts that will be covered in the thesis will be discussed, so that any reader, even less experienced, can deepen these topics and acquire a more solid basis to face the rest of the thesis without gaps and be able to understand it more lightly.

3.1 Formal Definition of Random Numbers

In order to explain what a Random Number is, we can mention the definition proposed by NIST Institute [10]: "A value in a set of numbers that has an equal probability of being selected from the total population of possibilities and, in that sense, is unpredictable. A random number is an instance of an unbiased random variable, that is, the output produced by a uniformly distributed random process."

3.1.1 The Continuous Uniform Distribution

One of the fundamental properties satisfied by random numbers is that they follow a uniform distribution. In probability theory , the continuous uniform distribution or rectangular distribution is a family of symmetric probability distributions. The distribution describes an experiment where there is an arbitrary outcome that lies between certain bounds. The bounds are defined by the parameters, a and b, which are the minimum and maximum values. The interval can either be closed (e.g. [a, b]) or open (e.g. (a, b)). Therefore, the distribution is often

abbreviated U (a, b), where U stands for uniform distribution [11]. The difference between the bounds defines the interval length; all intervals of the same length on the distribution's support are equally probable. This distribution represents the maximum entropy probability distribution for a random variable X under no constraint other than that it is contained in the distribution's support.

In probability theory, the Continuous Uniform Distribution is defined as a rectangular distribution since it allows to draw out an element over a uniform area, such that all the elements have the same probability of being drawn out. The rectangular area is delimited by 2 parameters which define the bounds of the rectangle, 'a' and 'b': by changing these parameters we define the underlying area of the rectangle. The distribution is often abbreviated as $U(a, b)$, where U stands for uniform distribution.

The corresponding Probability Density Function (PDF) is given by:

$$f(x) = \begin{cases} \frac{1}{b-a} & for\, a \leq x \leq b \\ 0 & for\, x < a\, or\, x > b \end{cases} \tag{3.1}$$

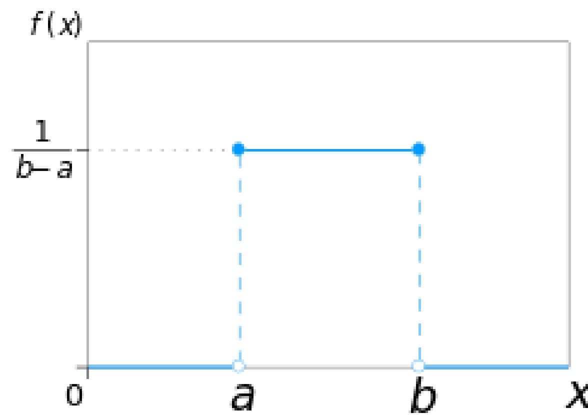

Figure 3.1: Probability Density Function of the Uniform Distribution

3.2 Random Number Generators

High quality Random Numbers are nowadays of vital importance for security-critical environments: indeed, in order for a cryptographic system to be defined as secure, it must use keys, seeds or nonces (i.e. sequences of bits necessary to encrypt a message or to exchange keys between two parties) that are completely random and therefore cannot be predicted. In fact, if it

would have been possible to predict such sequences, this would mean not only that they could no longer be used (because of the lack of secrecy) but also that the entire cryptographic system would be compromised.

3.2.1 Main features

As mentioned in [12], there exist different features that must be taken into account in order to understand how well an RNG device is behaving, in particular:

- **Uniformity**: Generated numbers should be evenly distributed in a given range;
- **Independence**: the numbers generated must not have correlations among them;
- **Period length**: The period must be high before a repeating sequence begins;
- **Replicability**: the generation of a sequence of numbers must be replicable (e.g. to allow for testing);
- **Speed**: clearly the algorithm must have a suitable speed for the various applications;
- **Controlled** memory usage;
- **Security**: In sensitive applications, such as encryption for online communications, it should not be possible to predict the result of the algorithm.

Each of these characteristics affects the quality and reliability of the numbers generated in output: for example, with the same quality of bits, it will always be preferred a faster generator than a slower one, or the one more resistant to attacks, or again the one which will have fewer times a sequence repeated in the output.

3.2.2 Different Random Number Generators in commerce

Random Number Generators (RNGs) are software or hardware that allow the production of Random Numbers. There exist mainly three different types of RNGs:

- **Pseudo-Random Number Generators (PRNGs)** Are based on an algorithm whose initial value is determined by a seed. PRNGs are able to produce output numbers that seem to be random but, due to the underlying deterministic algorithm that produces them, they are actually easily predictable. That said, over time it becomes extremely easy for a smart user to predict which sequences will be obtained from a PRNG's output, which is why currently PRNGs are no longer in use in cryptographic systems;

- **True Random Number Generators (TRNGs)** Base their strength on the use of unpredictable physical phenomena, such as avalanche noise, thermal noise, or clock drift [13]. Thanks to their non-deterministic nature, TRNGs offer an easy and cheap way to generate random data: the necessary physical components needed for randomness that must be installed on the TRNG hardware are all easily accessible, both from the point of view of availability and cost. While this can be seen as an advantage, it also appears to be the biggest drawback: in fact, being easy to find these materials, it is at the same time very easy to tamper with the hardware and compromise the use of the device;

- **Quantum Random Number Generators (QRNGs)** Are generators that base their randomness on the laws of quantum physics (which, by nature, are non-deterministic processes): according to the Born's rule [14], the possible outcome of a number cannot be predicted except at the exact moment in which it is measured. There are several methodologies for the generation of quantum random numbers: the greatest effort, given its large-scale implementation, is employed in trying to generate ever smaller and faster devices, while maintaining an affordable price.

4
The OneRNG TRNG Device

4.1 Device Design

OneRNG [15] is an open source random number generator, created by two developers based in New Zealand: Jim Cheetham and Paul Campbell. The device's design has gone through a number of incremental improvements, with versions 2 and 3 being made available for purchase via an associated e-shop. Due to the open source nature of OneRNG, it is easy to study both the hardware and software components of the device to get a deeper understanding of its functionality. In this chapter, the design of this device is more thoroughly examined.

As shown in Figure 4.1, OneRNG is designed to be connected to a host machine via a USB-A port. It is compatible with a number of operating systems such as various Linux distributions, MacOS and Windows.

OneRNG generates random data through two entropy sources that, namely an *avalanche diode* and a *channel-hopping radio receiver*, which are covered in more detail below.

4.1.1 Avalanche Diode

By applying a specific reverse bias voltage to a diode that is higher than it should receive, the forced passage of current allows free electrons (which acquire a certain kinetic energy) to hit a lattice of atoms of the diode present in the depletion region. This impact (which is commonly referred to as "avalanche breakdown" [16]) causes new electrons to detach from the lattice and,

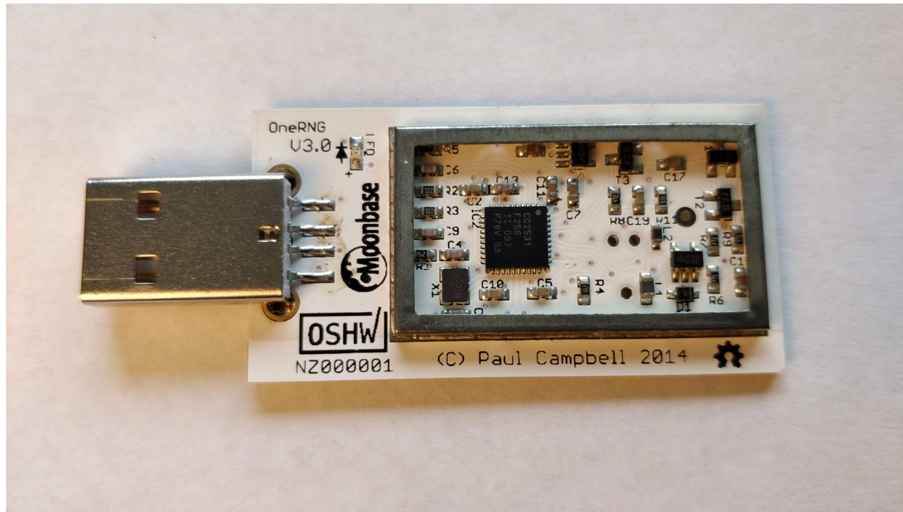

Figure 4.1: OneRNG Device

following a similar process, to hit other atoms, which will release even more electrons, generating a new reverse passage of current within the diode. This process can be used as a source of entropy as the exact current flowing across an avalanche diode while it is in avalanche breakdown is unpredictable, thus leading to an (almost) completely random process.

In Figure 4.2 it is possible to see the functioning of the avalanche diode: the positive charge attracts the free electrons (green dots) while the negative charge attracts the protons (red holes) leaving a large depletion region (yellow area): by applying an high voltage in input, the depletion region is filled with many electrons generating a passage of current.

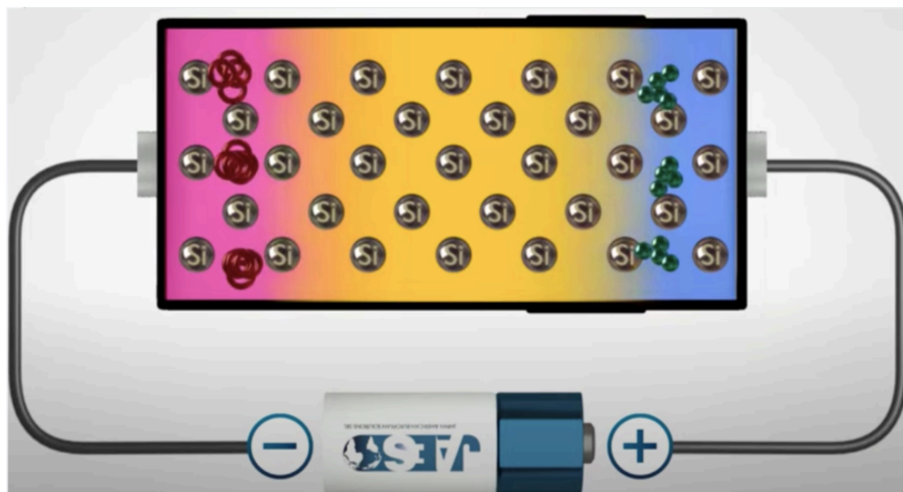

Figure 4.2: Avalanche diode in reverse bias conditions

4.1.2 Channel-Hoppin Radio Receiver

The OneRNG's second source of randomness uses an random number generation method that is pre-built into the CC2531 chip. This method measures RF signals from a randomly selected frequency (between 2394-2457MHz), and records the least significant bit. After 80-111 sampling loops, a new frequency is selected.

4.1.3 Data Whitening

In order to offset slight biases in the entropy generation methods (Approximately 5% more 1's than 0's in the avalanche diode system, and approximately 0.14% using the RF source), OneRNG also provides the option to perform data whitening on any collected random data before providing it to the host. Data whitening is a process that attempts to take data that may be initially correlated, and produce almost completely uncorrelated output. This is possible thanks to a whitening process that transforms a vector of random variables with a specific covariance matrix into a set of new variables whose covariance is the identity matrix: in particular, the covariance matrix (whose aim is to give information about how much data in input are correlated among them) projects input data along a specific direction. After the whitening process data, which now have identity covariance and variance equal to 1, are spread and projected into a sphere about the origin, thus deleting the dependencies from the initial direction (that is why the whitening transformation is also referred to as *sphering transformation*). Data whitening is used across different RNG devices, since it is a simple process that allows to eliminate any kind of bias or correlation among data.

The OneRNG implements data whitening by inputting the generated entropy into a CRC algorithm provided by the on board CC2531 chip. To do this, OneRNG feeds any generated input into a linear-feedback shift register (LFSR) one byte at the time, which triggers a CRC-16 calculation to be performed using the polynomial $x^{16} + x^{15} + x^2 + x$. The upper byte of the CRC calculation is then added to the entropy pool.

The developers recommend that rather than using this whitening method, users should instead use OneRNG as an entropy source for any random number generators already provided by their operating systems.

Table 4.1: OneRNG modes of operation.
Source: www.moonbaseotago.com/onerng/theory

Command	Avalanche Disable	RF Enable	Whitener Disable	Function
cmd0	0	0	0	Avalanche noise with whitener
cmd1	0	0	1	Raw avalanche noise
cmd2	0	1	0	Avalanche & RF noise with whitener
cmd3	0	1	1	Raw Avalanche & RF Noise
cmd4	1	0	0	No noise
cmd5	1	0	1	No noise
cmd6	1	1	0	RF noise with whitener
cmd7	1	1	1	Raw RF noise

4.2 Device Usage

Users can interact with the device by sending commands via a terminal interface. These commands can be used to select which entropy source(s) should be active, whether to apply data whitening to generated data, or dump program data. The full list of commands that is shown below in Table 4.1.

Typically, however, the user will not interact with the device directly. Instead, users are encouraged to modify a configuration file provided as part of the installation process. Despite the RF source being less prone to bias, by default, the OneRNG uses the avalanche diode circuit (with whitening enabled) to generate entropy. According to the developers, this is due to the RF source being potentially more vulnerable to outside interference.

4.3 Data Collection

For this work, we needed to create a reliable and reproducible environment with which to test the OneRNG devices. In this section, the steps taken to install and extract data from the OneRNG device will be detailed, such that the environment may be re-created.

4.3.1 Testing Environment

All testing that was performed on version three of the OneRNG device. In all instances of data collection, the connected host machines were using Ubuntu 22.04 installed over a VirtualBox.

4.3.2 Software Installation

In order for the OneRNG to function as intended, users are required to install a set of software from the OneRNG website. Initially, we attempted to install version 3.6-1 of the software, made available on the e-shop website. However, due to the age of the software, a number of compatibility issues were encountered. One of the dependencies required by the software, `python-gnupg`, was not able to be installed via the original method. After investigating further, the forum Instead, we modified the provided ".deb" package to instead depend on `python3-gnupg`, which was possible to install. Additionally, to work with newer kernel updates, an updated script was provided on a forum provided by the developers. For our testing, we implemented these changes, and were successfully able to interact with the device. An "official release" of these changes was released on the main website in April 2022, as version 3.7-1. This version contains minimal differences when compared to our implementation, and can be used as a substitute.

4.3.3 Device Operation

The software installation can be split into three parts: a `udev` rule, a shell script, and a configuration file.

First, the `udev` rule is used to detect when a OneRNG device is plugged in. Once a OneRNG device is detected, the shell script is run to configure the device and perform any necessary validation. The behaviour of this script is influenced by the configuration file, which can be accessed by the user to modify various device settings.

4.3.4 Collection Process

For our tests, we ideally wanted to extract data directly from the device. This can be achieved by reading directly from the device via the `/dev/ttyACMX` file (where X is the index of the USB device). While this allowed us to collect large amounts of data, we also wanted to be able to switch the device's modes of operation.

To operate the device as closely to its intended usage as possible, to switch modes we would modify the configuration file to the new mode, then re-plug the device. In some cases, we were even able to re-plug the device when the power of the host's USB ports were software controllable. During this work, we developed a shell script that could be used to automatically collect data from connected OneRNG devices with various modes, which is provided below:

```
    #!/bin/bash

###############CONFIG################
device_id="1-1:1.0" # check /sys/bus/usb/drivers/cdc_adc
for options
gensize="1gb" # NOTE: also have to change the dd "count".
datasource=$1
######################################

remount_rng() {
echo "Remounting OneRNG, this will take about
 15 seconds."
echo -n $device_id | sudo tee -a /sys/bus/usb/drivers/cdc_acm/unbind
 1>/dev/null
sleep 3
echo -n $device_id | sudo tee -a /sys/bus/usb/drivers/cdc_acm/bind
 1>/dev/null
sleep 10
}

modifycmd() {
echo "Mode set to cmd$1."
sed -i -r "s/ONERNG_MODE_COMMAND=\"cmd.*\"/ONERNG_MODE_COMMAND=
 \"cmd$1\"/" /etc/onerng.conf
}

reset_rng() {
echo "Restoring original configuration"
mv onerng.conf.bak /etc/onerng.conf
remount_rng
}

echo "----------OneRNG file generator----------"
```

```
if [ "$EUID" -ne 0 ]
  then echo "Must be running as root to modify OneRNG Config"
  exit
fi

# List out original config and save it
echo "'Default' config variables:"
grep -E '^ONERNG_START_RNGD|^ONERNG_MODE_COMMAND|^ONERNG_AES_WHITEN|
^ONERNG_URANDOM_RESEED|^ONERNG_ENTROPY|^ONERNG_FEED_KERNEL|
^ONERNG_FEED_RATE' /etc/onerng.conf

echo -e "\nSaving backup of current config to local directory.\n"
cp /etc/onerng.conf ./onerng.conf.bak

echo -e "Setting stty to fetch raw data"
stty -F /dev/ttyACM0 raw -echo

echo -e "Gen Filesize: $gensize\n"

# Loop through various cmd modes.
for mode in 1 3 6 7
do
echo "###Creating file: oneRNG_${datasource}_cmd${mode}.rand"
modifycmd $mode
remount_rng

# Are we getting directly from the device, or via /dev/random?
if [ "$datasource" = "direct" ]; then
dd bs=1024 if=/dev/ttyACM0 iflag=fullblock count=1048575
  of=oneRNGraw_${datasource}_cmd$mode.rand status=progress
elif [ "$datasource" = "devrand" ]; then
echo "Waiting 30 seconds so that the new mode can feed into the
```

```
  entropy pool a few times."
sleep 30
dd bs=1024 if=/dev/random iflag=fullblock count=1048575
  of=oneRNGraw_${datasource}_cmd$mode.rand status=progress
else
echo -e "\nError, no datasource selected.\nUSAGE: gen.sh
  [datasource]\n---Datasources:\n'direct' to fetch directly
  from OneRNG\n'devrand' to fetch from /dev/random\n\n"
reset_rng
exit
fi

done
reset_rng
```

In particular, the *OneRNG* device produces data at a rate of 53kb/s. For the experiments we had to conduct, we first collected 100MB of data for each of the 7 modes (cmd0-cmd7) of the device and, each time we performed an attack, we again collected the same amount of data with the same modes to analyze how much they differed from each other. The only 2 modes whose we avoided to collect data from are modes 4 and 5: this is because those 2 commands do not use any of the entropy sources of our interest.

5
The ComScire PQ32MU QRNG Device

In this chapter we will analyze in more detail the QRNG device PQ32MU [17] sold by the ComScire company. This device seems to be extremely promising by exploiting a quantum property called Quantum Shot-Noise, which guarantees high reliability on the data generated in output.

5.1 Device Design

This QRNG device was designed in accordance with recommendations provided by NIST (SP 800-90), which provide entropy source specifications for random number generators. This generator guarantees the passing of every statistical test aimed at analyzing the randomness of a device. Furthermore, this device has been designed to generate at least 1TB of data. Last but not least, all ComScire models have internally several tests designed to measure the quality of the output data such as the p-value, the z-score or the auto-correlation: the device has an internal threshold sensor which, if exceeded, indicates that the output is no longer generating the data correctly, blocking its leakage to prevent security issues. In image 5.1 it is shown the internal circuit of the device.

This device is capable of generating data at a speed of 32Mb/s with an estimated error of +-0.005%. The value of the auto-correlation is minimal (<1 part per trillion) and the estimated entropy is around 0.99 bits per output bit. The device looks like a black aluminum box closed with an internal FPGA: this is done to prevent the formation of humidity and the thermal

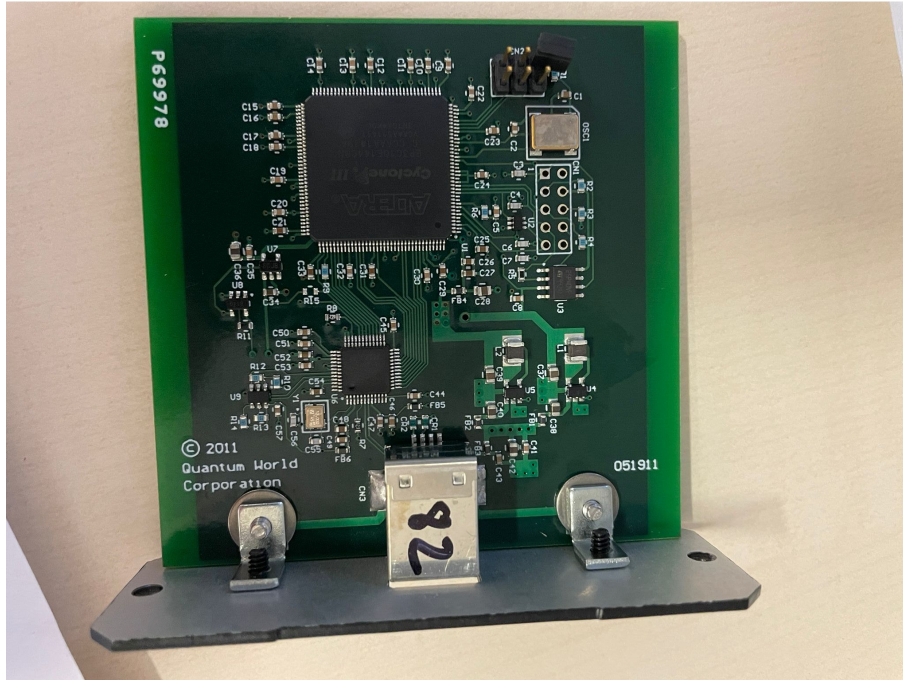

Figure 5.1: PQ32MU QRNG ComScire Device

conditions in which it can work are 0-50C°. The manufacturers suggest that it is suitable for gaming, security,cryptography and research. This QRNG device hosts an Altera Cyclone III FPGA, a low cost Integrated Circuit(IC) which also allows it to be programmed according to the specifications desired by the vendor. As entropy source, the QRNG PQ32MU uses a 12-LUT Ring Oscillator which allows the generation of quantum shot-noise - in the following sub-sections the entropy generation functioning is explained in detail.

5.1.1 Ring Oscillator

As shown in Figure 5.2, a ring oscillator [18] can be defined as a chain formed by an odd number of inverters (also called NOT gates) whose output oscillates between two voltage levels which represents the '0' bit (false) or the '1' bit (true). The last inverter output is connected in a feedback loop to the first inverter in input - that is why we refer to it with the term 'ring'. In particular, inverters are electronic components that given in input a Direct Current (DC) generate in output an Alternate Current (AC) with a square wave form: this is possible thanks to an oscillator which is able to turn on and off several transistors (which are often substituted by MOSFETs) at an high speed.

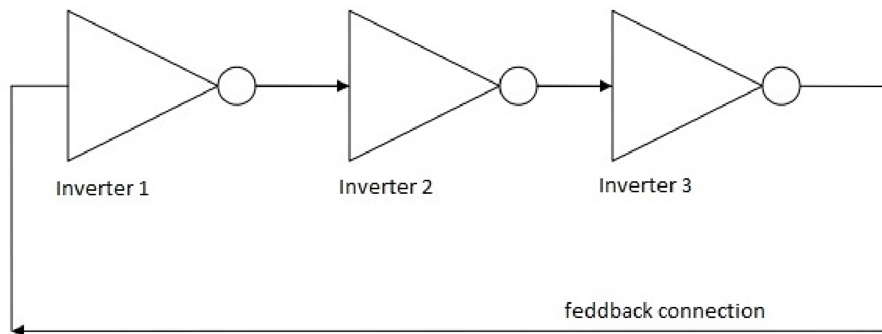

Figure 5.2: Ring Oscillator Diagram

Since the first inverter generates the logical NOT, by putting in series an odd number of NOT gates, also the last output generates again a logical NOT. The final output is asserted a finite amount of time after the first input is asserted and the feedback of the last output to the input causes oscillation. The number of ring oscillator's stages depends on the number of inverters N: if we refer to the ring-oscillator in figure 5.2, a 3-stage ring oscillator contains N=3 inverters in the chain.

Generation of the square wave

In order to understand how a ring oscillator works, first we need to understand what is a gate delay. In order to work properly, an inverter made with MOSFETs must first be charged and, after a specific amount of time, it generates the corresponding output: this generates a delay between the input and the output and, by adding several inverters in series, the delay increases even more causing a reduction in the frequency produced in output.

Given these premises, suppose to be in the situation in which an inverter has the same voltage in input and output and to have a small amount of noise in input to the inverter: this small amount of noise generates a voltage gain greater than 1 in output, which is then inverted and given in input again to the inverter through the feedback loop causing the square wave in output.

5.1.2 The Quantum-Shot Noise

The quantum-shot noise [19] is a noise that, due to its nondeterministic nature, can be defined as a quantum process and it is the source entropy of the QRNG ComScire PQ32MU. Similarly to the TRNG OneRNG device, the Altera Cyclone III FPGA contains a MOS [20] (Metal-Oxide Semiconductor) which works in reverse bias conditions: when in reverse bias, if in input the MOS receives a voltage higher than allowed, the p-n junction capacitance of

the diode increases generating a gate leakage of electrons which, by hitting the lattice of atoms, generates fluctuations of uncorrelated, quantized charge carriers which is by nature non deterministic.

The jitter

The jitter [21] can be generalized as a delay that may arise for many causes, such as thermal noise or time variations. In a n-LUT ring oscillator, at each n stage the total jitter increases as soon as we pass through each stage (each of the n inverters). The total cumulative jitter from these sources is the jitter introduced by a single stage multiplied by the square root of the number of stages the transition has passed through before being measured. In our case, the jitter is indeed caused by source entropy generated by the the quantum shot-noise signal in input, and it is possible to measure the amount of entropy generated by sampling in output the free-running oscillator with another oscillator. The Altera Cyclone III FPGA contains three connections which are equally-spaced along the 12-LUT ring oscillator. These three connections are then combined in a 3-input XOR gate to produce an enhanced ring oscillator output signal at three times the ring oscillation frequency: this allows also the generation of three independent entropy sources because the time spacing between the three connections is very large compared to the jitter distribution at each tap (over 10,000 standard deviations), and therefore the amount of mutual entropy due to sampling of overlapping jitter distributions is insignificant. Basically, first it is given in input to the MOS a slightly higher voltage than allowed which permits the generation of discrete quantize carriers; the amount of voltage generated in output is sampled and measured by the 12-LUT ring-oscillator and it is finally converted from analog to digital in order to return the stream of bits in output.

5.2 Device Usage

The ComScire device is extremely easy to install under Windows and relatively easy to install under a Linux OS, in particular we have installed the latest software version, v3.6. While on the Windows operating system the software contains an intuitive and easy to use GUI, on the Linux operating system all operations are performed and displayed from the terminal. As far as installation is concerned, ComScire has an online guide which explains in detail all the steps to be taken to install the software depending on the operating system we are using. In particular, in our case we used Ubuntu 22.04 installed on the VirtualBox v7.0 virtual machine.

There are 3 libraries to install:

- **LIBUSB-1.0** which allows the device to be recognized by the USB port;

- **LIBFTDI1** which, used together with LIBUSB-1.0 allows a connection to talk to the FTDI's UART/FIFO chips integrated in the device;

- **LIBQWQNG**, a library that allows access to the device (only after changing the access privileges to the .sh file with the command chmod +x ubuntu-x64.sh.

After installation, you can verify that everything went well by running several tests listed in the *./examples* folder:

$ ./examples/clear

$ ./examples/deviceid

$ ./examples/diagnostics

$ ./examples/errorhandl

$ ./examples/randbytes

$ ./examples/randint32

$ ./examples/randnormal

$ ./examples/randuniform

$ ./examples/reset

$ ./examples/runtimeinfo

Once installation is complete, the device is connected to the computer via a USB2.0 cable supplied in the box: the device automatically starts generating data and at the same time providing information about the tests it carries out: *p-value, z- score, entropy and auto-correlation*.

5.3 Data Collection

For our purposes, we have generated an executable file `./lotta` which allows the generation of a specific amount of data which are automatically saved on file: in fact, all the statistical tests are performed not by directly analyzing the data generated in output but by inputting the test the `.rand` file directly.

We first collected 10 files of 100MB each and at least 1 file of 100MB for each type of attack performed. The executable file `./lotta` allows to decide the amount of data to generate and save it on a file whose we can choose the name:

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <qwqng.hpp>

QWQNG* QNG;

// Author: Calvin Brierley (crb34@kent.ac.uk)
// Based upon original example code provided by ComScire (RandBytes.cpp)
int main(int argc, char *argv[])
{
    if(argc != 2) {
    printf("Incorrect number of arguments.\n");
    printf("Expected usage: %s [number of bytes]\n", argv[0]);
    return EXIT_FAILURE;
    }

    char* p;
    long totalBytes = strtol(argv[1], &p, 10);
    if(!*p == 0) {
    printf("Provided argument was not a number, exiting.\n");
    return EXIT_FAILURE;
    }



    int qngStatus = 0;

    QNG = new QWQNG(); // create class

    //printf("\n%s\n\n", "Start Device...");

    /* Print Status String */
    char* statusString;
    statusString = QNG->StatusString();
    //printf("Status: %s\n\n", statusString);
```

```
    /* Print open device serial number */
    char* SerialNumber;
    SerialNumber = QNG->DeviceID();
    //printf("DeviceID %s\n\n", SerialNumber);



    /*  Loop through and get bytes */
    //printf("\nGetting %s random bytes\n\n", argv[1]);
    char* randbyte;
    int bytecount;
    while(totalBytes > 0){
    if(totalBytes >= 1024) {
    bytecount = 1024;
    } else {
    bytecount = totalBytes;
    }

    randbyte = new char [bytecount];
    if ( (qngStatus = QNG->RandBytes(randbyte, bytecount)) != S_OK) {
printf("Error: %s\n\n", QNG->StatusString());
return EXIT_FAILURE;
    }
    for (int i=0; i<bytecount; i++) {
printf("%c", randbyte[i]&0xFF);
    }
    totalBytes -= bytecount;
    //printf("\nBytes left to print: %ld\n\n", totalBytes);
    }

    // Cleanup
    delete [] randbyte;
    delete QNG;
```

```
    //printf("\n\n%s\n\n", "EXIT...");

    return EXIT_SUCCESS;
}
```

6
Statistical Tests

In order to measure the reliability of the output of OneRNG and ComScire devices, several state-of-the-art statistical tests have been chosen: their goal is precisely to perform different types of tests on the data generated in the output to verify if there is any kind of correlation between the measured bits. It is legitimate to wonder why different statistical tests are carried out on the output data instead of choosing only one that we deem the most accurate: the reason is because each of the tests taken into consideration perform a different type of analysis on the data, thus allowing not only to check whether the output is random or not, but also whether the possible internal correlation of the data is for example long-range or short-range, or whether the output fails on a specific test. All this information is extremely important to be able to carry out an in-depth analysis on the quality of the output and understand how, after performing a specific attack, the data has been modified.

We have considered five statistical tests, in particular: Ent, Booltest, PractRand, TestU01 and BitReps, each with different characteristics to allow us to analyze different aspects on the obtained data - a detailed analysis of them follows in this chapter.

6.1 Ent

Ent [22] is a straightforward and easy-to-use statistical test for testing randomness: it performs 5 different sub-tests on an input file and gives in output information about if the input stream can be defined as random or not.

Ent computes five different tests which are defined as follows:

- **Entropy:** measures the information density of a file content, basically analysing its compression: for a byte file, the entropy value should be a value near to 8 while for a bit file, this value should be near 1;

- **Chi-square test (χ^2)** : is an extremely sensitive test and give precise information about if a file is truly random or not. Firstly, Chi-square test computes the expected number of occurrences in a given bit sequence and then compares this value with the one obtained by the output file: if the difference between the expected uniform value and the output file is large, it is possible to conclude that the file contains signs of non-randomness. For a byte output file, the Chi-square value can be computed as:

$$\chi^2 = \sum_{i=0}^{255} \frac{(x_i - E[x_i]])^2}{E[x_i]]}$$

where x_i is the number of occurrences of byte i obtained from the output file while $E[x_i]$ is the expected number of occurrences under the assumption of a uniform output;

- **Aritmetic Mean**:

It gives information about the bytes average value for a byte output file this mean should be around 127.5 and can be easily computed as:

$$\bar{x} = \sum_{i=0}^{n-1} \frac{x_i}{n};$$

- **Montecarlo value for π**: a sequence of 6 bytes is used to compute the 24-bit X and Y coordinates within a square. For almost random sequences, the inscribed circle in this square should assume a value near to π ;

- **Serial correlation coefficients**: measures how much a byte depends on the previous one. For completely random data, this value should be near to 0; for pseudo-random data instead this value is nearly 0.5 while if data are completely correlated, the serial correlation coefficient value may reach values near to 1.

Moreover, *Ent* allows to specify 5 different options, each of which allows you to make a specific analysis on the data:

- **[-b]**: the input file is read as a stream of bits instead of bytes;
- **[-c]**: prints the number of occurrences for each byte (or bit if -b option is enabled);
- **[-f]**: folds upper-case letters to lower-case letters according to the ISO 8859-1 standard;

- **[-t]**: output file is in CSV format which allows for an easier analysis of data;
- **[-u]**: just prints the how-to-call information.

In the image below it is possible to see the output of *Ent* test: the first row indicates the different parameters that *Ent* takes into consideration while the corresponding results for that particular data file are shown in the second row. By specifying the `-t` parameter, we require that the output is also shown in `.svg` format: indeed, the third row shows the amount of `'0'`s and `'1'`s contained in the data file and the corresponding percentage of appearance for each of them.

```
ludovica@ludovica-VirtualBox:~$ ent -b -c -t  RNG-battery-tests/Ent/ComScire_1.b
in
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,3.524778,0.499967,3.141545,-0.000035
2,Value,Occurrences,Fraction
3,0,400026551,0.500033
3,1,399973449,0.499967
```

Figure 6.1: Ent statistical test output

6.2 BoolTest

BoolTest [23] is based on looking for Boolean functions that exhibit an unexpected bias and identifying the simplest function that does so.

The Monobit test , which looks at the ratio of ones and zeros in the given sequence, served as the model for BoolTest. When in Monobit test, $f(x_1) = x_1$ is applied to all of the sequence's bits, its outcomes is testing whether number of ones '1's and '0's are close to each other as would be expected for random data.$f(x_1, x_2, ..., x_m)$ of m variables can be used to apply any Boolean function on non-overlapping blocks of m bits.

Additionally, in order to find connected bits in the bit level processing, we must repeatedly run the process with different relations and bit chosen, which is actually time consuming. However, Booltest simply needs only a brief amount of time to evaluate the test. The performance was verified on more than 20 real world cryptographic functions – block and stream ciphers, hash functions and pseudorandom generators.

In Booltest, input data are divided into multiple non-overlapping blocks and then the polynomials are built based on the bit values depicted in a block. After finishing the analysis of the input data, Booltest computes the distribution of bit-level polynomials based on the given

input and compares them with the expected distribution of truly random data. The level of randomness of a given data is computed by checking a value known as ***z-score***: if the *z-score of observed distribution* is outside the interval of the *z-score of predefined expected truly of random data*, the tested data is considered to be non-random, otherwise it is reported as acceptably random.

Z-score and P-value

Z-score is a numerical measurement, which follows the standard normal distribution of the original data, and performs value's relationship to the mean of a group of value. It measures the distance between the mean using standard deviation and a certain data point. *Z-score* can be positive or negative. It is positive if the observed value is above the mean, and it is negative if observation value is below the mean. In Booltest the stronger distinguisher (Boolean function) is obtained by the bigger value of z-score.

P-value is a probabilistic indicator of how closely perfect random sequence (expected value) and random sequence being tested (observed value) match. Instead of computing *p-value*, one can directly compute *z-score*, since they are related and *z-score* is also simpler and faster.

In the image below a possible output of BoolTest is shown. The analysis is done through the *z-score* value: if the output from data lies in the interval `[4.775841 , 7.609938]`, the randomness test is passed and Booltest output the log "BoolTest could not find statistically significant non-randomnes".

```
2023-01-11 16:26:35 ludovica-VirtualBox booltest.booltest_main[3789] INFO Proces
sing finished in 17.806448936462402 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0
}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: ComScire_1.bin
The best distinguisher found: (x_{19} * x_{83}) + (x_{20} * x_{168})
 - z-score achieved: 5.385500304088172

BoolTest has reference statistics for used configuration, number of collected sa
mples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.385500304088172 lies in the allowed interval, the rand
omness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 6.2: BoolTest statistical test output

6.3 TestU01

TestU01 [24] is an extremely powerful software library implemented in ANSI C which allows to perform a great variety of statistical tests on data and consequent analysis of the results.

It is possible to use this test not only with your own random number generators (for example, specific RNG hardware) but also with some already predefined ones in the library, which represent the best known generators proposed in the literature or found in widely-used software. The same goes for statistical tests: TestU01 proposes some of the most famous ones already existing in the literature but also most recent ones. It provides general implementations of the classical statistical tests for random number generators, as well as several others proposed in the literature, and some original ones. It gives also the option to use basic tools for plotting vectors of points produced by generators.

Additional software permits one to perform systematic studies of the interaction between a specific test and the structure of the point sets produced by a given family of random number generators. That is, for a given kind of test and a given class of random number generators, to determine how large should be the sample size of the test, as a function of the generator's period length, before the generator starts to fail the test systematically.

This library consists of four modules:

1. **'u' module**: to implement RNGs;
2. **'s' module**: to implement single statistical tests;
3. **'b' module**: to implement pre-defined battery of tests, which consist of different single statistical tests defined in 's' module;
4. **'f' module**: implementing tools for applying tests to entire families of generators.

The name of every library element, such as type, variable or function, is prefixed by the name of the module to which it belongs. Chapters 2 to 5 of the User's guide describe these four classes of modules and give some examples.

TestU01 comes with a comprehensive guide[25] which helps the user to perform different analysis according to his specific needs. For our needs, we wrote a simple C program which takes as a generator in input directly the data files generated by OneRNG and ComScire devices and executes specific tests on these data. The program performs the Rabbit battery test, since together with the Alphabit battery tests are the only two batteries whose it is possible to decide

the amount data to give in input. Here follows the code, where everytime it is necessary to specify the data file name which in this case is the "ComScire.bin" file:

```
#include "unif01.h"
#include "bbattery.h"
#include "gdef.h"
#include "swrite.h"
#include <stdio.h>

int main(void)
{

swrite_Basic = FALSE;
bbattery_RabbitFile ("ComScire.bin", 100000000)
return 0;

}
```

In the figure below it is shown a summary of the *Rabbit* battery test applied on the ComScire.bin file: after a brief summary about the version in use, the type of generator and the time used by the CPU to compute the test, the TestU01 output shows only the singular statistical tests (if any) belonging to the battery that have not been passed and the corresponding p-value obtained in output.

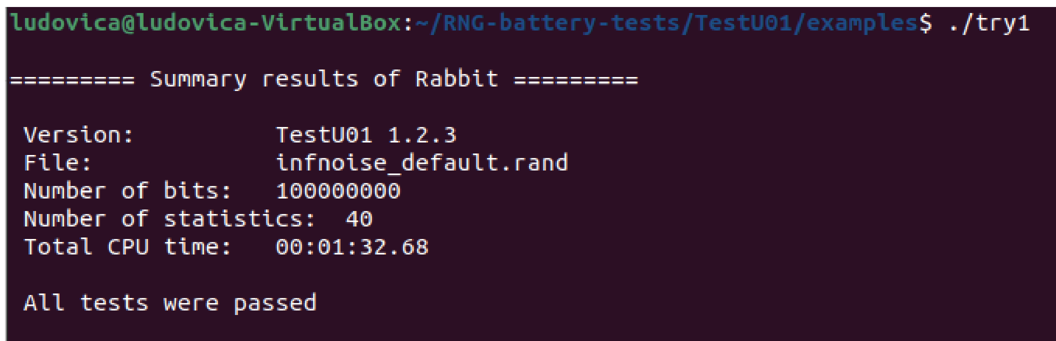

Figure 6.3: TestU01 Rabbit statistical test output

6.4 PractRand

PractRand [26], which stands for *'Practically Random'*, is a C++ library that works both as an RNG and as a Statistical Test for analyzing the randomness of data. With the provided commands, it is possible to give in input both the files produced by the PractRand RNG or, if needed, also new input files. Simlar to what *Ent Test* does, *PractRand* performs a variety of sub-tests, this time trying to find out both correlations on long and short range sequences: in fact in many cases the different attacks on RNGs are made by making sure that analyzing the data over short range (i.e. based on short data sequences), they seem unrelated but once it is performed a longer range analysis (i.e. based on very long sequences) it is possible to notice a strong dependence among output data.

The five subtests that *PractRand* performs are presented above:

- **BCFN**: checks for long range linear correlations (bit counting), in practice this test looks for Fibonacci style repetitions that, relying on long sequences, often escape other statistical tests;
- **DC6**: checks for short range linear correlations (bit counting) and this is done basically by looking for overlapping values of short sequences;
- **Gap16**: counts the number of digits that appear between repetitions of a particular digit and then uses the Kolmogorov-Smirnov test to compare with the expected number of gaps
- **BRank**: sequences of bits are given in input as rows to a matrix, then it is checked for linear dependencies among different rows;
- **Float Point Frequency - FPF**: checks for short range correlations (shorted than DC6 test)

As for the interpretation of the results, *PractRand* progressively updates the analysis status in output showing the RNG name, the number of bytes tested, the time taken and the RNG seed used. Once the test has analyzed all the input data, if one or more anomalies have been detected an output table is shown containing the name of the test on which the sequence has "failed" (i.e. it could not be judged as random), the corresponding p-value of the data found on that test and a final evaluation that indicates whether the sequence seems only *"suspicious"* (i.e. signs of non-randomness have been detected but they are not enough to declare the sequence as compromised for that test) or if it is completely compromised and so it is not possible to use

it anymore. On the contrary, if the test does not detect anomalies, it ends as soon as all the bytes have been analyzed and does not show any message, unless the "-p 1" option is enabled: in that case the output table will show all the results obtained from the different tests and will mark them as *"normal"* .

```
length= 512 kilobytes (2^19 bytes), time= 10.1 seconds
  no anomalies in 84 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 12.1 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 14.5 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 16.6 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 18.7 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 20.9 seconds
  no anomalies in 151 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 23.3 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 25.9 seconds
  no anomalies in 179 test result(s)
```

Figure 6.4: PractRand statistical test output

In figure 4.4 it is possible to see part of the *PractRand* statistical test output. Not surprisingly, this test analyzes increasing bit sequences, each time increasing the power it is observing by 2: in this way it can first apply the short-range tests on short bit sequences and then pass to an analysis on longer sequences.

PractRand is a very lightweight and powerful test, it is extremely accurate and tries to solve the problem raised in *Ent*: at the expenses of a large use of memory, by being able PractRand to obtain information on both short-range and long-range sequences, it ranks among the most promising tests currently in use.

6.5 BitReps

Developed by the University of Kent as part of the Quantum Communications Hub, Bitreps [27] is a statistical test for identifying signs of non-randomness within the output of an RNG. By making use of the Bloom filter, its purpose is to identify bit-level repetitions in a sequence: this is done by comparing the observed level of repetition with the expected level of repetition that a random sequence should have, basically measuring the distinguishability between two probability distributions.

6.5.1 Mathematical Theory

The aim of BitReps is to compute the expected number of genuine repetitions for a given RNG output. Given a set $\{1, ..x\}$, let X_n denote the number of distinct results if the set $\{1, ..x\}$ is drawn n times.

Then the expected number of genuine repetitions is given by:

$$E[X_n|X_{n-1} = r] = \frac{r}{x} \cdot r + \left(1 - \frac{r}{x}\right) \cdot \left(r + 1\right) = \left(1 - \frac{1}{x}\right) \cdot r + 1$$

which can be rewritten as:

$$E[X_n|X_{n-1}] = \left(1 - \frac{1}{x}\right) \cdot X_{n-1} + 1$$

So the expected value of X_n can be computed as:

$$E[X_n] = E[E[X_n|X_{n-1}]] = 1 + \left(1 - \frac{1}{x}\right) \cdot E[X_{n-1}]$$

which, solving the recursive relation, results in:

$$E[X_n] = 1 + \left(1 - \frac{1}{x}\right) + \left(1 - \frac{1}{x}\right)^2 + \left(1 - \frac{1}{x}\right)^3 + .. + \left(1 - \frac{1}{x}\right)^{n-1}$$

that, finally, can be rewritten as:

$$E[X_n] = x\Big(1 - \Big(1 - \frac{1}{x}\Big)\Big)^n$$

Let b represent the blocksize (in bits) of a given sequence: the output of an RNG can be considered the sampling of the set {x=0, ..., 2^b}. Using the above formula, BitReps calculates the expected number of genuine repetitions for a given RNG output, being n the number of blocks. In practice, in order to compute the expected number of repetitions of a given sequence, BitReps makes use of Bloom Filters.

6.5.2 The Bloom Filter

BitReps bases its theory on Bloom filters [28], designed in the 1970s by developer Burton Howard Bloom. These filters are extremely useful because they allow to work with large amounts of data while maintaining high efficiency in terms of memory used, speed and computational complexity. A Bloom filter can be described as an approximate data structure since it is able, in a very short time, to provide information about whether one or more elements are (probably) present in a set or if they certainly are not: in practice, the final result may be a false positive but for sure it will not be a false negative.

In this section, after briefly explaining the basic properties of hash functions, we are going to see how Bloom Filters work, what makes them so efficient and how do we exploit them in our work.

- **Hash Functions**

  A hash function is a function $h : X \rightarrow Y$ that maps any arbitrary long input X to a fixed length output Y. In particular, this function has two properties:

  - for any given input X to an hash function, the corresponding output $Y = h(X)$ is easy to compute but hard to invert: while it is easy to compute Y knowing X, it is hard knowing Y to retrieve information about X.

  - Given the distribution of the input X, the output distribution Y is a uniform distribution over the set of the possible outputs: $Y \sim U(Y)$

- **How the Bloom Filter works**

  Given a set of hash functions $\{h_1, h_2..h_n\}$, an array of elements starting from index 0 and a set of inputs $\{x_1, x_2..x_n\}$, first of all every input is hashed with each hash function present in the set (e.g. for input x_1 we need to compute all the hash functions

$\{h_1, h_2..h_n\}$): the corresponding outputs $\{y_1, y_2..y_n\}$ obtained by each single input are represented as integer values that indicate the corresponding output storage location in memory (e.g. suppose that the hash function h_1 of input x_1 assumes value 4 in output, then it will be stored in the array in position 4).

Once this operation is performed for all the input values, the obtained array of elements will contain a '1' in the indices where an hash function for a given input is computed, a '0' otherwise.

That said, it is important to mention that different inputs for different hash functions could assume the same output values: given two different inputs x_1 and x_2 , it is likely that the output obtained (so the value we are going to save in the array of elements) is the same for the two inputs even if the hash functions that generated those outputs are different.

This is precisely the crucial point of the Bloom filter: when we want to know if an element is present in the array or not, first we give it in input to the Bloom filter, then the filter will compute the corresponding hash functions for that element and finally it will check if the corresponding outputs are present in the array or not. As soon as an output is not present, it means that surely that element cannot be present in the array, while if all the outputs are present, it means that it is likely that the element is present but, given that such output value could be assumed also by other inputs, there is no absolute certainty that the found element is exactly the one we asked for.

- **Efficiency**

  Bloom filter efficiency arises from two main factors:

  1. The computational complexity required to perform the hashing operations on the input and verify the presence (or not) of the element in memory is polynomial.
  2. Little memory space is required in order to store all the output values: being m the Bloom Filter's length, memory usage is $\frac{m}{8}$ bytes plus a few bytes of overhead.

- **How BitReps makes use of Bloom filters**

  Given an RNG that emits sequences of bits at a certain speed, our aim is to monitor whether these sequences are independent and uniform and therefore can be defined as completely random. The use of Bloom filters helps in this sense, since by applying this filter to the sequences in output from the RNG, BitReps can declare whether a certain sequence has already (probably) been observed previously in output or not: if the answer is affirmative, it means that the output is not homogeneous as specific sequences of bits may (probably) have been repeated.

6.5.3 Main Features

BitReps is implemented through a Graphic User Interface (GUI) which greatly simplifies its use. As soon as the GUI opens it gives the possibility to select two types of screens: the *Calculator* and the *Analyzer*.

Calculator

Gives information about the eventual number and location of repetitions obtained from RNG output. As shown in 6.5 , once in this section, it is possible to choose the file on which computing repetitions by clicking on the *Select Input Data* button. Then it is required to fill in all the remaining field:

- **Block size**: the number of bits over which a repetition is measured. It is possible to choose among six different fixed value: 8, 16, 32, 64, 128 and 512 bits;
- **Sliding Window**: if selected, shifts the blocks by one bit a time instead of by the chosen block size;
- **Error rate:** the maximum desired error rate for the underlying Bloom Filter.

In the absence of files to monitor, BitReps provides a file that can be immediately used to test data: it is located in the input folder, the RNG that produced it is provided by a PRNG found on Unix-like operating systems called `/dev/urandom` and the file name is `urandom100M-2.bin`

Once the parameters have been set, by clicking on the `Run` button BitReps starts to compute all the calculations: a progress bar is shown providing information on the amount of data analyzed and the time taken to perform the analysis.

As soon as BitReps finishes monitoring all bits, a `.json` file is written in the `output` folder with the name:

```
[filename of RNG data]-blocksize
-errorrate-slidingwindow.json
```

which will then be necessary for the analysis section.

Finally, it is possible to click on the `reset` button to reset all inputs and allow the user to select a new file.

Analyzer

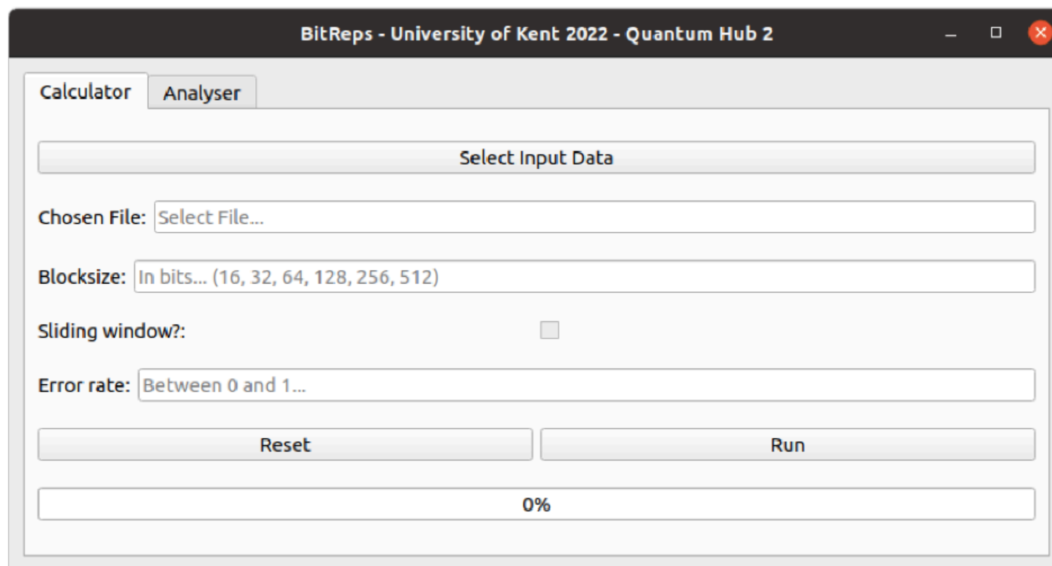

Figure 6.5: BitReps Calculator Window

Performs a statistical analysis on the repetitions that have been found. 6.6 shows the Analyzer section: firstly, by clicking on the `Select Input Data` button it is possible to select the analysed file which can be found in the `output` folder; then, by clicking on the `Select Model` button it is possible to choose a reference model that will be needed to perform a comparison between a supposed to be uniform data distribution and the actual output that has been obtained from the `Calculator` section.

Once data file has been selected, by clicking on the `Run` button, an automated analysis is performed. On the left-side screen it is possible to visualize the parameters that have been chosen for analysing the file whilst in the right-side screen are shown the different values necessary to perform the analysis on the data:

- **Expected distribution:** The expected distribution used in the chi-square calculation
- **Observed distribution:** The observed distribution used in the chi-square calculation
- **Chi-square**: The chi-square value of the observed vs. expected distribution
- **Expected false positives:** The expected number of false positives given the size of the Bloom filter
- **Expected duplicates:** The expected number of genuine repetitions given the blocksize and size of the RNG output

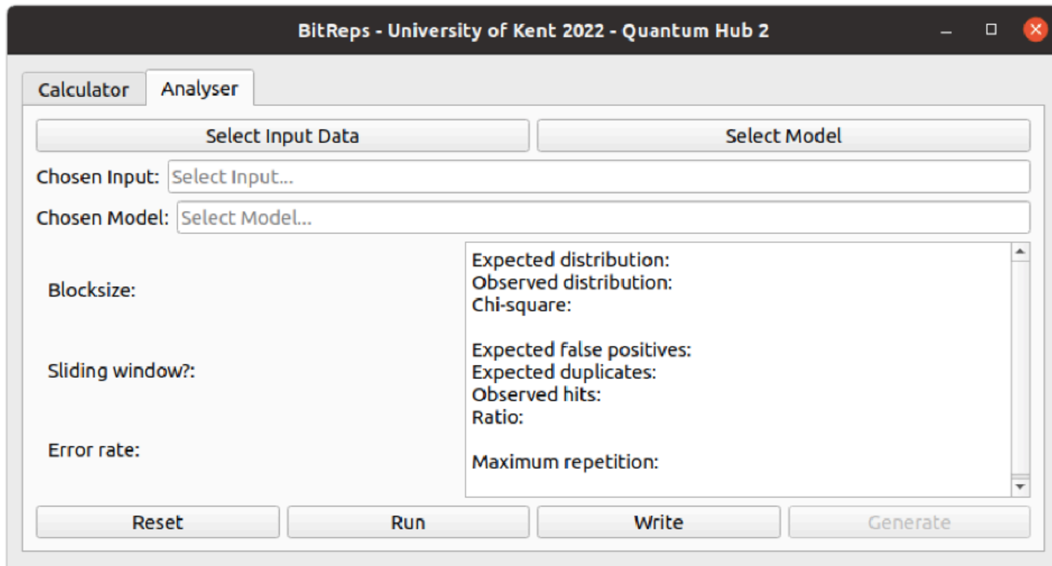

Figure 6.6: BitReps Analyzer Window

- **Observed hits:** The number of repetitions measured by BitReps

- **Ratio**:

$$\frac{\text{N}^\circ \text{ expected repetitions}}{\text{N}^\circ \text{ observed repetitions}} \tag{6.1}$$

- **Maximum repetition:** The single most repeating ”word” in the RNG output (as well as its binary representation)

The `Write` button can be used to write the information from the right-hand side of the GUI to a file in the directory using the same naming convention as for the `output` file.

The `Generate` button (reserved for a future release) can be used to generate various graphical visualisations of the repetition within given RNG output with respect to the frequency and period of repetition.

The `Reset` button can be used to reset the metadata as well as the results obtained from automated analysis.

6.5.4 Result interpretation

In figure 6.7 it is possible to see two different outputs obtained by two different files: the first one defines the analysed file as random while the second one defines the file as compromised (non-random).

```
Random Data -

Expected distribution: [80141, 178]
Observed distribution: [79487, 164]
Chi-square: 6.44

Expected false positives: 20
Expected duplicates: 79837
Observed hits: 79651
Ratio: 1.0

Maximum repetition: 2 (11000011000010111011101000110100)
```

```
Non-Random Data -

Expected distribution: [80141, 178]
Observed distribution: [1742239, 581907]
Chi-square: 1935643441.75

Expected false positives: 20
Expected duplicates: 79837
Observed hits: 3177644
Ratio: 0.03

Maximum repetition: 3016 (00000000000000000000000000000000)
```

Figure 6.7: BitReps statistical test output

In the random file, it is possible to see that the observed distribution stays in the same interval of the expected distribution: that is why the Chi-square has a low value. On the contrary, in the non-random file the observed distribution is much more larger that the expected one, making the Chi-square value extremely high.

Scrolling through the various values obtained, another important data is provided by the analysis of the *Ratio* which can be obtained by dividing the expected number of duplicates by the observed number of duplicates: a low *Ratio* means too many repetitions of specific sequences compared to the ideal ones, resulting in a high probability that the sequence will be defined as non-random.

7
Temperature Attacks

In this section we will discuss the two attacks carried out: both attacks were carried out in the research laboratory of the University of Kent, as we needed hand and face protection and specific tools that could only be used in the laboratory. In particular, the first attack involves bringing both devices to very low temperatures while the second, through the use of a *hot air gun*, brings both devices to very high temperatures: in both cases the goal was to test how much these thermal changes affect the output of the devices.

7.1 The cooling attack

7.1.1 Setup

For this attack, as can be seen in the figure 7.1, we needed goggles and a protective mask as well as safety gloves. Furthermore, we purchased 12 different freezing sprays, as we had to replace them every time the device temperature fell below -30°C: the liquid inside was therefore sufficient for only one experiment at a time. In addition, we also used an infrared thermometer to constantly measure the temperature and a camera to record the whole experiment.

Figure 7.1: Setup of Cooling experiment

7.1.2 The attack

We started the attack starting from the ComScire QRNG device: after connecting the device via USB, we entered the command `./lotta 10000000 > Cooling_attack.rand` from the terminal: in this way we asked the device to generate 100MB of data and save them in a file called Cooling_attack.rand. While the data extraction was in progress, we aimed the freezing spray on the CPU contained within the device's hardware and, in the meantime, we monitored the temperature through the infrared thermometer. As soon as we reached the maximum temperature of -30°C, we terminated the experiment.

The same is valid for the OneRNG TRNG device: we modified some parameters inside the `main.py` file of the program that allowed us to extract a specific amount of data for each of the 7 possible modes of the device, setting again the amount of data to extract for each of the 7 ways equal to 100MB. At the end of this process, the OneRNG device data files were automatically generated in output adding the post-fix on each file name about the specific operating mode that was in use i.e. during the extraction data of operating mode 1 the filename in output was

OneRNG_raw_direct_cmd1.rand

Once this was done, and after starting the data extraction, we followed exactly the same process used for the ComScire device: we aimed the freezing spray nozzle at the hardware CPU and in the meantime we monitored the temperature through the infrared thermometer. As soon as we reached -30°C, we ended the experiment.

As it is possible to see in the figures 7.2 and 7.3, these were the devices after having suffered the attack: both devices were completely frozen, in particular, the temperature of the device is clearly lower near the CPU where a minimum temperature of -31°C is reached and slightly higher in the outer parts (-16°C).

Figure 7.2: ComScire device after Cooling experiment

Figure 7.3: OneRNG Device after Cooling experiment

7.2 The heating attack

7.2.1 Setup

The setup of the heating attack is similar to that of the freezing attack, i.e. first we connected one device at a time to the PC thanks to the USB input and then, on each of the two devices, we performed the attack while we were extracting data.

As it is possible to see in figure 7.4, for this attack we used a *hot air gun* which was available at the Shed of the University of Kent's Institute of Computer Science. This *hot air gun* emits continuous and linearly increasing heat, up to a maximum temperature of 550°C.

For the ComScire QRNG device we performed the experiment only once during the data extraction. The whole experiment lasted 3 minutes which is the amount of time needed by the device to generate data: during this amount of time we were pointing the *hot air gun* toward the CPU until it reached the maximum allowed temperature.

Figure 7.4: Setup of Heating experiment

For the OneRNG TRNG device instead we repeated the experiment at fixed intervals of 10 minutes for each of the 7 operating modes. The whole experiment for this device lasted around 3 hours: this is because each operating mode from 1 to 7 requires around 30 minutes (53kb/s) to generate 100MB of data. Again, during this amount of time the *hot air gun* was pointing toward the CPU until it reached the maximum allowed temperature. The extraction method remains the same discussed in the Cooling attack:

- For the QRNG ComScire device,we entered the command
  ```
  ./lotta 10000000 > Heating_attack.rand
  ```
  from the terminal: in this way we asked the device to generate 100MB of data and save them in a file called Heating_attack.rand;
- For the TRNG OneRNG device, we let run the `main.py` program that allowed the extraction of 100MB of data from each of the 7 operating modes of the device.

While we were performing the attack and we were pointing the *heating gun* towards the CPU of the devices, we also measured constantly the temperature with the infrared thermome-

ter such that it was possible to monitor which was the maximum temperature that could be reached by the two devices.

Unlike the previous attack, for this experiment it was necessary to cover both devices with a thermal tape and leave only the CPU part uncovered, allowing to concentrate the heat only in that area and to keep the temperature on the other components unchanged: this was done to prevent the heat from damaging the remaining parts of the device, as although the CPU is able to withstand very high temperatures, on the contrary there are some components (such as resistors) that stop working at high temperatures thus requiring the need to protect such areas.

7.2.2 The attack

While the data was being extracted, we aimed the *hot air gun* again near the CPU of both devices: as soon as we reached the maximum temperature, we moved the *hot air gun* to a point outside the device to try to keep the heat in equilibrium at a fixed temperature as far as possible.

While for the ComScire device we performed the heating attack only once (the output rate is high and 100MB of data was produced in 3 minutes), for the OneRNG device, having an output rate significantly slower than the ComScire (53kb/s, therefore a data extraction time of about 30 minutes), we repeated the experiment at regular intervals of 10 minutes for each of the 7 operating modes: as soon as the device reached room temperature, the experiment was repeated and the whole experiment lasted, as in the first attack, around 3 hours in total.

In figures (7.5 and 7.6) it is possible to see the status of the devices during the attack.

In particular, we made several tests to understand which was the ideal maximum temperature that the two devices were able to reach, concluding that:

- for the ComScire QRNG device, the maximum temperature it manages to reach without compromising the device is 170°C: it was possible to draw this conclusion because as soon as we tried to exceed this temperature, the device activated a protection mechanism for which it terminated the extraction of data, allowing the quality of the data produced up to that moment to remain stable;
- for the TRNG OneRNG device, the maximum temperature reached is 80°C: we were able to ascertain this result because, once this threshold was exceeded, the LED that usually flashes while it is extracting entropy, gradually stopped working as the temperature rose, meaning the device has stopped generating entropy.

Figure 7.5: Heating Attack on TRNG OneRNG device

Figure 7.6: Heating Attack on QRNG ComScire device

8
Results Analysis

In this chapter we will analyze all the results obtained and make a comparison between the performances reached by the TRNG OneRNG device compared to those obtained by the QRNG PQ32MU device.

In Section 8.1 are analyzed, using the four statistical tests reported in Chapter 6, the data files obtained from the two devices. The data files were all collected as described in the "Data Collection" paragraphs 4.3 and 5.3.

In Section 8.2 instead are analyzed the data files collected while the devices were subjected to the temperature attacks following the same methodology used in section 1.

Section 8.3 will follow, aimed at carrying out an accurate analysis on the results obtained, also making an in-depth study depending on the type of test carried out: in fact, we want to try to understand not only if the attack was successful or not, but also in what how the attack affected the data. For example, the data might be more predictable over long-range rather than short-range sequences; or there may be a strong auto-correlation value on specific bit sequences; or again simply the amount of '0' or '1' used could be completely unequal, perhaps with 30% '0' and 70% '1'.

Section 8.4 is instead entirely dedicated to the analysis of the results obtained by testing the data with the BitReps statistical test. This statistical test was entirely written with Python programming language by the PhD student Jamie Pont at the University of Kent (UK) and has been the subject of in-depth analysis, since it has never been tested before.

The output results obtained with each statistical test is shown for OneRNG device with

modes 0,1,3,6 and 7 of the OneRNG device as specified in section 4.2 and, since the ComScire PQ32MU device has only 1 operating mode, only one result is shown for it. We avoided to use modes 4 and 5 of the OneRNG since those operating modes do not use the channel-hopping Radio Frequency (which is the main mode used in order to generate entropy), meaning that the output has a quite low entropy and so it is not useful for our goal (e.g. OneRNG would not give in output an interesting result when under attack using those modes).

8.1 Results obtained with devices in normal conditions

Ent

First of all, we are going to see the results obtained by using Ent Statistical Test on data files collected when both devices are in normal conditions. In Figure 8.1 it is possible to see the resulting output: the options enabled for Ent were [-b -c -t], so that the output is shown in bits and not in bytes and the output is in `.svg` format. Through the [-c] option, we specify that we want to print in output the number of occurrences for each bit.

Besides cmd1 and cmd3 which return in output and entropy value respectively of 0.80 and 0.94, for all the other OneRNG operating modes and for ComScire device the entropy obtained is of 1 or at most 0.99, meaning that the numbers are completely random.

BoolTest

The second statistical test we have used is BoolTest with the following common testing parameters:

```
--top 128 --no-comb-and --only-top-comb --only-top-deg --no-term-map
--topterm-heap --topterm-heap-k 256
```

As it is possible to see from figure 8.2 to figure 8.7, BoolTest it is an extremely precise and scrupulous test, in fact only the data obtained from the ComScire device and the OneRNG device with modes 6 and 7 were defined as random, while the OneRNG files in modes 0, 1 and 3 were defined as non-random files.

This result is already interesting in itself as it shows how it is not enough to analyze data with a single statistical test but there is a need to compare the results obtained with different tests so that it is possible to judge with certainty whether a certain file is random or not.

TestU01

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/OneRNG/oneRNGraw_direct_cmd0.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,838860800,1.000000,1.375629,0.500020,3.141308,-0.000009
2,Value,Occurrences,Fraction
3,0,419413415,0.499980
3,1,419447385,0.500020
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/OneRNG/oneRNGraw_direct_cmd1.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,838860800,0.805663,215649906.616544,0.246487,3.945600,0.022571
2,Value,Occurrences,Fraction
3,0,632092252,0.753513
3,1,206768548,0.246487
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/OneRNG/oneRNGraw_direct_cmd3.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,838860800,0.941621,66968891.419727,0.358726,3.748743,0.070173
2,Value,Occurrences,Fraction
3,0,537939448,0.641274
3,1,300921352,0.358726
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/OneRNG/oneRNGraw_direct_cmd6.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,838860800,1.000000,3.054118,0.500030,3.141438,0.000000
2,Value,Occurrences,Fraction
3,0,419405092,0.499970
3,1,419455708,0.500030
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/OneRNG/oneRNGraw_direct_cmd7.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,838860800,1.000000,0.165875,0.500007,3.140773,0.000021
2,Value,Occurrences,Fraction
3,0,419424502,0.499993
3,1,419436298,0.500007
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t rng-data/ComScire.bin
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,2.628695,0.499971,3.141644,0.000015
2,Value,Occurrences,Fraction
3,0,400022929,0.500029
3,1,399977071,0.499971
```

Figure 8.1: OneRNG and Comscire data files analyzed with Ent Statistical Test

For the TestU01 Statistical test we experimented data using the Rabbit battery, since together with the Alphabit battery tests are the only two batteries whose it is possible to decide the amount data to give in input. Here follows the code, where everytime it is necessary to specify the data file name which in this case is the "ComScire.bin" file:

```
#include "unif01.h"
#include "bbattery.h"
#include "gdef.h"
#include "swrite.h"
#include <stdio.h>

int main(void)
{

swrite_Basic = FALSE;
bbattery_RabbitFile ("ComScire.bin", 100000000)
```

```
2023-02-07 10:49:48 ludovica-VirtualBox booltest.booltest_main[3965] Processing finished in 17.46407723426819 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: ComScire.bin
The best distinguisher found: (x_{66} * x_{82}) + (x_{189} * x_{242})
 - z-score achieved: 5.751232913153013

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.751232913153013 lies in the allowed interval, the randomness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.2: ComScire data file analyzed with BoolTest Statistical Test

```
2023-02-07 10:51:35 ludovica-VirtualBox booltest.booltest_main[3969] Processing finished in 19.95348286628723 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: OneRNG/oneRNGraw_direct_cmd0.rand
The best distinguisher found: (x_{199} * x_{201}) + (x_{8} * x_{229})
 - z-score achieved: 30.783148820451856

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 30.783148820451856 lies outside of the interval, the randomness hypothesis is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.3: OneRNG cmd0 data file analyzed with BoolTest Statistical Test

```
    return 0;

}
```

From figure 8.8 to figure 8.11 it is possible to see the obtained results when applying Rabbit Battery test to the 5 OneRNG modes files and to the ComScire file.

The reason the results obtained using modes 1 and 3 of the OneRNG device are missing is because the test was unable to finish the computation: this happens when the data files have such low entropy that it would take up too much memory and calculation work to measure the actual values in output. The OneRNG mode 6 and 7 data files and the ComScire data file passed all output tests, meaning no non-random statistical data was noted. On the contrary, the OneRNG cmd0 data file also shows all the tests that have not been passed and the relative p value obtained that goes indeed outside the parameters.

PractRand

From figure 8.12 to figure 8.17 are shown the obtained results when applying PractRand statistical test to the 5 OneRNG modes files and to the ComScire file. PractRand doesn't signal if a specific data file has passed the tests or not, just shows for each string of data taken in input if all the test for that block size were passed or not. As already shown for BoolTest statistical test, also in this case ComScire and OneRNG modes 0, 6 and 7 passed all tests while OneRNG

```
2023-02-07 10:52:46 ludovica-VirtualBox booltest.booltest_main[3973] INFO Processing finished in 16.012020349502563 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: OneRNG/oneRNGraw_direct_cmd1.rand
The best distinguisher found: (x_{116} * x_{121}) + (x_{115} * x_{120})
 - z-score achieved: 1005.7532640415384

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 1005.7532640415384 lies outside of the interval, the randomness hypothesis is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.4: OneRNG cmd1 data file analyzed with BoolTest Statistical Test

```
2023-02-07 10:53:44 ludovica-VirtualBox booltest.booltest_main[3977] INFO Processing finished in 19.993804454803467 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: OneRNG/oneRNGraw_direct_cmd3.rand
The best distinguisher found: (x_{0} * x_{5}) + (x_{1} * x_{6})
 - z-score achieved: 856.7852803754093

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 856.7852803754093 lies outside of the interval, the randomness hypothesis is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.5: OneRNG cmd3 data file analyzed with BoolTest Statistical Test

modes 1 and 3 data files returned several tests that were not passed. In particular, were refused the tests for the BCFN, DC6 and FPF parameters, which represent the parameters linked to short range sequences: while on long data sequences the data result to be completely random, this hypothesis fails when analyzing short sequences.

8.2 Results obtained with devices under attack

In this section are analyzed the data files when both devices are under attack. In the subsections 8.2.1 and 8.2.2 are reported the results obtained respectively when devices are under the cooling attack and the heating attack. Each of the data files of both devices are again analyzed with each statistical test. Taking into consideration the results obtained by analyzing data files when they are under normal conditions, we avoid to repeat the attack on the data files OneRNG with modes 1 and 3 since they didn't pass the majority of statistical tests already in normal conditions.

```
2023-02-07 10:57:28 ludovica-VirtualBox booltest.booltest_main[3982] INFO Processing finished in 18.14283776283264 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: OneRNG/oneRNGraw_direct_cmd6.rand
The best distinguisher found: (x_{169} * x_{171}) + (x_{63} * x_{216})
 - z-score achieved: 6.209804495714829

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 6.209804495714829 lies in the allowed interval, the randomness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.6: OneRNG cmd6 data file analyzed with BoolTest Statistical Test

```
2023-02-07 10:58:24 ludovica-VirtualBox booltest.booltest_main[3987] INFO Processing finished in 19.539316654205322 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: OneRNG/oneRNGraw_direct_cmd7.rand
The best distinguisher found: (x_{8} * x_{252}) + (x_{90} * x_{153})
 - z-score achieved: 6.1230817574431535

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 6.1230817574431535 lies in the allowed interval, the randomness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.7: OneRNG cmd7 data file analyzed with BoolTest Statistical Test

8.2.1 The cooling attack results

ComScire PQ32MU results

In Figures from 8.18 to 8.21 are shown the results obtained when analyzing the ComScire data file with the four statistical tests when the device is under the cooling attack. As it is possible to see, all the tests were passed meaning that when bringing the ComScire device to very low temperatures (-30°C), this does not compromise in any manner the output result.

OneRNG results

From figures 8.22 to 8.28 are shown instead the results obtained when the OneRNG device is under the cooling attack. As it is possible to see and differently from the data reported when the device is under normal conditions. OneRNG cmd0 data file is considered not random from all the statistical tests but Ent, whose entropy measured is 0.9965, which is still a good entropy value. OneRNG cmd6 and cmd7 remain quite stable since the entropy in output is still near to 1. PractRand result for OneRNG cmd0 data file fails on almost every FPF test, meaning that the data file is rich of repetitions on the short range. It was not possible to collect any result from Rabbit battery test when analyzing the OneRNG cmd0 data file. The results

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             rng-data/ComScire.bin
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:41.00

 All tests were passed
```

Figure 8.8: ComScire data file analyzed with TestU01 Rabbit Battery Statistical Test

obtained for cmd6 and cmd7 OneRNG data files are the same (no anomalies in all the tests), that is why their results are reported in the same image. This meaningful result tell us how the cooling attack did not influence almost at all both devices, since all the results remained almost the same both when devices are in normal conditions and under the cooling attack.

8.2.2 The heating attack results

In this section are analyzed the results obtained when the devices are under the heating attack. The process about how we performed this attack are reported in section 7.2.

ComScire results

For the ComScire device, we first made several experiments in order to understand which was the maximum temperature that the device could reach keeping working properly. After some trials, we concluded that the maximum temperature admissible is of 180°C. We could verify this result since when we tried to exceed this temperature (reaching 200°C), the device stopped to generate random numbers: indeed we are going to analyze the results obtained when the heating gun reached 180°C and those obtained when are reached 200°C. For the last case, since the device stopped its functioning when 200°C were reached, the data file size is of 77.1MB instead of 100MB as for all the others.

Again, as it is possible to see from figure 8.29 to figure 8.32the ComScire device even under the heating attack returns very good results: indeed it passed all the tests of all statistical tests and its the entropy is still 1.0 even when we reach 200°C.

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ ./try1

========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             rng-data/OneRNG/oneRNGraw_direct_cmd0.rand
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:31.37
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  MultinomialBitsOver              eps
  6  LempelZiv                      1 - eps1
  9  LongestHeadRun                  4.8e-8
 11  HammingWeight                    eps
 15  HammingIndep, L = 16             eps
 16  HammingIndep, L = 32             eps
 17  HammingIndep, L = 64             eps
 20  Run of bits                      eps
 24  RandomWalk1 H                    eps
 24  RandomWalk1 M                    eps
 24  RandomWalk1 J                    eps
 24  RandomWalk1 R                   1.3e-5
 25  RandomWalk1 M (L = 1024)       1.1e-10
 ----------------------------------------------
 All other tests were passed
```

Figure 8.9: OneRNG cmd0 data file analyzed with TestU01 Rabbit Battery Statistical Test

OneRNG results

In this section, from figure 8.34 to figure 8.40, the results obtained when the OneRNG device is under heating attack are reported. The results are extremely significant, as it is possible to see how this time, unlike the others, all data files are classified as non-random by any test. In particular, looking at the figure 8.34, we see that the amount of '0' and '1' is decidedly disproportionate with 80% of '0' and 20% of '1'. The Rabbit battery belonging to the TestU01 statistical test does not produce any results when analyzing these files, that's why the results obtained from this test have not been reported.

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             rng-data/OneRNG/oneRNGraw_direct_cmd6.rand
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:02:18.72

 All tests were passed
```

Figure 8.10: OneRNG cmd6 data file analyzed with TestU01 Rabbit Battery Statistical Test

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             rng-data/OneRNG/oneRNGraw_direct_cmd7.rand
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:02:13.49

 All tests were passed
```

Figure 8.11: OneRNG cmd7 data file analyzed with TestU01 Rabbit Battery Statistical Test

8.3 Result Analysis

Summarizing what was done in this chapter, we first analyzed the data files generated by both devices when they were under normal conditions, then we performed the same analysis on the data files generated by the two devices when they were under attack. All data files, where possible, were analyzed with all four statistical tests mentioned in the 6 chapter. In particular, the only test that sometimes did not produce results was the Rabbit Battery of TestU01: this is because the analysis carried out by this test is so thorough and requires so many computational calculations and memory that if the file was non-random, does not allow the test to complete the operation.

As it has been possible to see from the results reported, the ComScire device is able, when it suffers this type of attack, to react promptly: in fact, as soon as it is no longer able to guarantee the randomness of the data produced, the device automatically stops data generation. On the contrary, this does not happen in the OneRNG device which, when it underwent the heating

```
rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 11.3 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 13.4 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 15.4 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 17.4 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 19.6 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low4/32]Gap-16:B                 R=  -4.2  p =1-1.0e-3   unusual
  ...and 150 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 21.9 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 24.8 seconds
  no anomalies in 179 test result(s)
```

Figure 8.12: ComScire data file analyzed with PractRand Statistical Test

attack, simply reduced the output entropy compromising the generated data.

This result is extremely important, as a random number generation device should be as resilient as possible when external conditions change, and react like the ComScire device when these conditions are not met, thus stopping data generation to avoid compromise the system.

While we can't add much more regarding the analysis of the results obtained by the ComScire device (as it passed every test under every attack performed), we can carry out a more in-depth analysis of the results obtained by the OneRNG device. We recall that the cooling attack did not produce interesting results, as on that occasion the OneRNG device was able to withstand low temperatures without compromising the data generated: in fact, the results obtained are practically identical to those obtained when the device was under normal conditions. On the contrary, really significant results were obtained when the OneRNG device was subjected to the heating attack, which we analyze below.

The first interesting thing to notice is how the entropy has dropped from a value close to 1.0

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/PractRand$ cat rng-data/OneRNG/oneRNGraw_direct_cmd0.rand | ./RNG_test stdin -tlmin 1KB
RNG_test using PractRand version 0.95
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 1 kilobyte (2^10 bytes), time= 0.1 seconds
  no anomalies in 6 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 0.9 seconds
  no anomalies in 25 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 kilobytes (2^14 bytes), time= 1.6 seconds
  no anomalies in 30 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 kilobytes (2^15 bytes), time= 2.3 seconds
  no anomalies in 45 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 kilobytes (2^16 bytes), time= 3.9 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R= +17.8  p =  5.3e-9    VERY SUSPICIOUS
  ...and 53 test result(s) without anomalies
```

Figure 8.13: OneRNG cmd0 data file analyzed with PractRand Statistical Test

to a value of 0.56 (cmd0 file) in the worst case and 0.72 in the best case (cmd7 file). Furthermore, referring to the Ent test shown in figure 8.34it is curious how the quantity of '0' and '1' always tends to be with a ratio of 80/20, therefore 80% of '0' and 20% of '1'. The Chi-square value is extremely high, 431557441.22 in the worst case and 284019462.63 in the best case. Observing instead the results obtained with BoolTest and shown in figures 8.35, 8.36 and 8.37 , the z-score reaches a peak of 1246.34 in the worst case: bearing in mind that the possible interval is [4.77 , 7.60], this is definitely not a good result .

As regards the results obtained with the PractRand statistical test, it is possible to notice how all three OneRNG data files show almost the same output: the failed tests concern analyzes of short sequences, meaning that the data files are full of short repeated sequences (e.g 010101, 111000 repeated many times within the whole data file)

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/PractRand$ cat rng-data/OneRNG/oneRNGraw_direct_
cmd1.rand | ./RNG_test stdin -tlmin 1KB
RNG_test using PractRand version 0.95
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 1 kilobyte (2^10 bytes), time= 0.1 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  -2.7  p = 0.967     unusual
  ...and 5 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 1.0 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R=+111.9  p =  4.4e-27    FAIL !!
  DC6-9x1Bytes-1                    R=+119.5  p =  2.6e-51    FAIL !!!!
  Gap-16:B                          R=  +5.8  p =  1.6e-3   unusual
  FPF-14+6/16:cross                 R= +26.3  p =  4.2e-17    FAIL !
  [Low1/8]DC6-9x1Bytes-1            R= +33.1  p =  4.9e-10    VERY SUSPICIOUS
  [Low4/32]DC6-9x1Bytes-1           R= +26.0  p =  4.4e-8     VERY SUSPICIOUS
  ...and 19 test result(s) without anomalies
```

Figure 8.14: OneRNG cmd1 data file analyzed with PractRand Statistical Test

8.4 BitReps

This section is entirely dedicated to the analysis of the results obtained through BitReps. Its operation is fully explained in the ?? chapter. Since this is still a primitive version of BitReps, it is not extremely efficient, i.e. the computation time required to analyze a 100MB data file ranges from 10 minutes to 40 minutes, depending on the block size (e.g. for a 32-bit block size, the time required is 40 minutes while for a block size of 512 bits the time required is 10 minutes). The analysis in this case was carried out only on the data files under normal conditions, this is because if we try to carry out the analysis on the data files compromised by the attacks, BitReps cannot contain all the operations in the RAM memory, blocking and terminating the process.

The results have been reported in the tables below.

We performed the test on:

- the data files extracted from the OneRNG device with cmd0, cmd1, cmd3, cmd6 and cmd7 modes;

- the data files extracted from another TRNG called Infnoise [29].

We also analyzed the same file with each of the five modes, each error rate present in the

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/PractRand$ cat rng-data/OneRNG/oneRNGraw_direct_
cmd3.rand | ./RNG_test stdin -tlmin 1KB
RNG_test using PractRand version 0.95
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 1 kilobyte (2^10 bytes), time= 0.2 seconds
  no anomalies in 6 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.5 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 1.0 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R= +79.3  p =  1.7e-19    FAIL !!
  DC6-9x1Bytes-1                    R= +28.1  p =  2.1e-12    FAIL
  FPF-14+6/16:all                   R= -15.4  p =1-6.2e-8   very suspicious
  FPF-14+6/16:cross                 R= +13.8  p =  2.3e-9     VERY SUSPICIOUS
  [Low1/8]DC6-9x1Bytes-1            R=  +7.5  p =  5.6e-3   unusual
  ...and 20 test result(s) without anomalies
```

Figure 8.15: OneRNG cmd3 data file analyzed with PractRand Statistical Test

range [0.00001, 000.1 , 00.1, 0.1] and every possible block size (BS) of those proposed by the software (16, 32, 64, 128, 256, 512).

When we tried to analyze the file with a BS=16 bits, BitReps stopped its operation at about 56% of the progress bar: it was not possible to report any results for this BS for any of the error rates. The results obtained for each error rate vary minimally, for this reason and for ease of reading we report only the tables with error rate = 0.00001.

As it is possible to see from the tables, the smaller the BS, the more precise the output result: in fact, when we analyze the data with BS=32, the chi-square generated in the output is extremely sensitive even to the slightest variations (e.g. a chi- square very high for minimal repetition of bit sequences).

The tables show all the data that can be viewed in the BitReps Analyzer window. The last column is marked with ✓ if the file is random, with x otherwise. When we parsed the data files with BS=32 bits, the only file that could not be computed was the OneRNG_cmd1.rand file. As regards the analysis of the results obtained with BS=32 bits, it is possible to deduce that the files Infnoise_raw, oneRNG_cmd0, oneRNG_cmd1, oneRNG_cmd2 and oneRNG_cmd3 are not random: this confirms the analysis made by the other statistical tests (Ent, PractRand, BoolTest and TestU01). For the non random files the chi-square value is really high while the

	Observed distribution	Chi-square	Observed hits	Ratio	Maximum repetition	Random
Infnoise_default	[79508, 164]	2.23	79672	1.0	2	✓
Infnoise_raw	[953269, 27415]	14699420.49	981314	0.08	4	x
OneNRG_cmd0	[652205, 61002]	29493114.2	726560	0.11	9	x
OneNRG_cmd1	-	-	-	-	-	-
OneNRG_cmd2	[146503, 914]	60558,29	147442	0.54	3	x
OneNRG_cmd3	[1275934, 223225]	358861808.24	1611055	0.05	54	x
OneNRG_cmd6	[79790, 153]	1.54	79944	1.0	3	✓
OneNRG_cmd7	[79473, 161]	1.54	79635	1.0	3	✓

Table 8.1: Bitreps, block size=32 bits, error rate= 0.00001, expected distribution [79480, 146] , expected duplicates 79837, Expected false positives 20

	Observed distribution	Chi-square	Observed hits	Ratio	Maximum repetition	Random
Infnoise_default	[6]	0.5	6	1.67	1	✓
Infnoise_raw	[9]	0.12	9	1.11	1	✓
OneNRG_cmd0	[11]	1.12	11	0.91	1	✓
OneNRG_cmd1	[22]	24.5	22	0.45	1	x
OneNRG_cmd2	[10]	0.5	10	1.0	1	✓
OneNRG_cmd3	[13]	3.12	13	0.77	1	x
OneNRG_cmd6	[10]	0.5	10	1.0	1	✓
OneNRG_cmd7	[10]	0.5	10	1.0	1	✓

Table 8.2: Bitreps, block size=64 bits, error rate= 0.00001, expected distribution [8] , expected duplicates 0, Expected false positives 10

	Observed distribution	Chi-square	Observed hits	Ratio	Maximum repetition	Random
Infnoise_default	[7]	0.8	7	0.71	1	x
Infnoise_raw	[6]	0.2	6	0.83	1	✓
OneNRG_cmd0	[6]	0.2	6	0.83	1	✓
OneNRG_cmd1	[7]	0.8	7	0.71	1	x
OneNRG_cmd2	[4]	0.2	4	1.25	1	✓
OneNRG_cmd3	[4]	0.2	4	1.25	1	✓
OneNRG_cmd6	[3]	0.8	3	1.67	1	✓
OneNRG_cmd7	[3]	0.8	3	1.67	1	✓

Table 8.3: Bitreps, block size=128 bits, error rate= 0.00001, expected distribution [5] , expected duplicates 0, Expected false positives 5

	Observed distribution	Chi-square	Observed hits	Ratio	Maximum repetition	Random
Infnoise_default	[-]	0.0	1	3.0	1	✓
Infnoise_raw	[-]	0.0	2	1.5	1	✓
OneNRG_cmd0	[-]	0.0	4	0.75	1	✓
OneNRG_cmd1	[-]	0.0	3	1.0	1	✓
OneNRG_cmd2	[-]	0.0	3	1.0	1	✓
OneNRG_cmd3	[-]	0.0	3	1.0	1	✓
OneNRG_cmd6	[-]	0.0	2	1.5	1	✓
OneNRG_cmd7	[-]	0.0	4	0.75	1	x

Table 8.4: Bitreps, block size=256 bits, error rate= 0.00001, expected distribution [-] , expected duplicates 0, Expected false positives 3

	Observed distribution	Chi-square	Observed hits	Ratio	Maximum repetition	Random
Infnoise_default	[-]	-	-	-	-	-
Infnoise_raw	[1]	3.2	1	1.0	1	✓
OneNRG_cmd0	[1]	3.2	1	1.0	1	✓
OneNRG_cmd1	[-]	-	-	-	-	-
OneNRG_cmd2	[2]	1.8	2	0.5	1	✓
OneNRG_cmd3	[-]	-	-	-	-	-
OneNRG_cmd6	[-]	-	-	-	-	-
OneNRG_cmd7	[2]	1.8	2	0.5	1	✓

Table 8.5: Bitreps, block size=512 bits, error rate= 0.00001, expected distribution [5] , expected duplicates 0, Expected false positives 1

```
rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 11.3 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 13.2 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 15.4 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low1/8]BCFN(2+2,13-8,T)          R= +11.3  p =  8.4e-4   unusual
  ...and 123 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 17.6 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 19.8 seconds
  no anomalies in 151 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 22.1 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 24.7 seconds
  no anomalies in 179 test result(s)
```

Figure 8.16: OneRNG cmd6 data file analyzed with PractRand Statistical Test

ratio is really small. A completely random file should instead have a low chi-square and an high ratio near 1.

As soon as we increase the BS, the results get worst and worst up to BS=512 bits where all the results were '0'. This was really meaningful: indeed BitReps loses precision as soon as we increase the block size, meaning that it, or at least its 1.0 version is able to obtain concrete results only with BS=32 bits. On the other hand, we can also affirm how accurate this software is, as it is able to detect even the slightest error in a sequence: in fact, when the analysis is done using the BS=32 bits, BitReps is able to reveal very accurately whether the file is random or not.

```
rng=RNG_stdin, seed=unknown
length= 512 kilobytes (2^19 bytes), time= 9.3 seconds
  no anomalies in 84 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 11.2 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 13.2 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 15.2 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 17.2 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low1/8]DC6-9x1Bytes-1            R=  +6.3  p =  2.4e-3   unusual
  ...and 134 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 19.4 seconds
  no anomalies in 151 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 21.7 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 24.4 seconds
  no anomalies in 179 test result(s)
```

Figure 8.17: OneRNG cmd7 data file analyzed with PractRand Statistical Test

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/ComScire_A/Cooling$ ent -b -c -t ComScire_2_A_total.bin
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,0.171230,0.500007,3.141696,-0.000039
2,Value,Occurrences,Fraction
3,0,399994148,0.499993
3,1,400005852,0.500007
```

Figure 8.18: ComScire data file analyzed with Ent Statistical Test when the device is under the cooling attack

```
2023-02-07 22:07:40 ludovica-VirtualBox booltest.booltest_main[3037] INFO Processing finished in 21.432579755
78308 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: ComScire_2_A_total.bin
The best distinguisher found: (x_{91} * x_{177}) + (x_{186} * x_{200})
 - z-score achieved: 5.647238656901172

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.647238656901172 lies in the allowed interval, the randomness hypothesis cannot be r
ejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.19: ComScire data file analyzed with BoolTest Statistical Test when the device is under the cooling attack

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             ComScire_2_A_total.bin
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:41.81

 All tests were passed
```

Figure 8.20: ComScire data file analyzed with Rabbit TestU01 Statistical Test when the device is under the cooling attack

```
rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 15.9 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 18.1 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 20.4 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 22.6 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  +6.7  p =  1.3e-3   unusual
  [Low4/32]Gap-16:A                 R=  +5.0  p =  1.0e-3   unusual
  ...and 149 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 25.4 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 28.2 seconds
  no anomalies in 179 test result(s)
```

Figure 8.21: ComScire data file analyzed with PractRand Statistical Test when the device is under the cooling attack

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/oneRNGraw-direct-gen-A/Cooling$
 ent -b -c -t oneRNGraw_direct_cmd0.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,0.996594,3774762.784800,0.465655,3.316171,0.013927
2,Value,Occurrences,Fraction
3,0,427476400,0.534346
3,1,372523600,0.465655
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/oneRNGraw-direct-gen-A/Cooling$
 ent -b -c -t oneRNGraw_direct_cmd6.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,0.000381,0.500000,3.140986,-0.000041
2,Value,Occurrences,Fraction
3,0,399999724,0.500000
3,1,400000276,0.500000
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/oneRNGraw-direct-gen-A/Cooling$
 ent -b -c -t oneRNGraw_direct_cmd7.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,1.549856,0.500022,3.141623,0.000028
2,Value,Occurrences,Fraction
3,0,399982394,0.499978
3,1,400017606,0.500022
```

Figure 8.22: OneRNG data file analyzed with Ent Statistical Test when the device is under the cooling attack

```
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0
}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw_direct_cmd0.rand
The best distinguisher found: (x_{0} * x_{5}) + (x_{3} * x_{6})
 - z-score achieved: 210.58135806085025

BoolTest has reference statistics for used configuration, number of collected sa
mples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 210.58135806085025 lies outside of the interval, the ran
domness hypothesis is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.23: OneRNG cmd0 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack

```
2023-02-07 22:51:11 ludovica-VirtualBox booltest.booltest_main[3252] INFO Proces
sing finished in 16.937458515167236 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0
}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw_direct_cmd6.rand
The best distinguisher found: (x_{71} * x_{118}) + (x_{3} * x_{61})
 - z-score achieved: 5.321234190674209

BoolTest has reference statistics for used configuration, number of collected sa
mples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.321234190674209 lies in the allowed interval, the rand
omness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.24: OneRNG cmd6 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack

```
2023-02-08 08:52:45 ludovica-VirtualBox booltest.booltest_main[3883] INFO Proces
sing finished in 16.831845998764038 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0
}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw_direct_cmd7.rand
The best distinguisher found: (x_{46} * x_{115}) + (x_{78} * x_{133})
 - z-score achieved: 5.623869161114424

BoolTest has reference statistics for used configuration, number of collected sa
mples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.623869161114424 lies in the allowed interval, the rand
omness hypothesis cannot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.25: OneRNG cmd7 data file analyzed with BoolTest Statistical Test when the device is under the cooling attack

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ ./try1

========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             oneRNGraw-direct-gen-A/Cooling/oneRNGraw_direct_cmd6.rand
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:34.76

 All tests were passed



ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ gcc try1.c -o
 try1 -ltestu01 -lprobdist -lmylib -lm
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ ./try1

========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             oneRNGraw-direct-gen-A/Cooling/oneRNGraw_direct_cmd7.rand
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:35.51

 All tests were passed
```

Figure 8.26: OneRNG cmd6 and cmd7 data file analyzed with TestU01 Rabbit Battery Statistical Test when the device is under the cooling attack

```
rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  -4.3  p =1-3.2e-3   unusual
  ...and 11 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 0.9 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R= +31.8  p =  2.0e-8    VERY SUSPICIOUS
  DC6-9x1Bytes-1                    R= +10.0  p =  1.1e-4   mildly suspicious
  ...and 23 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 16 kilobytes (2^14 bytes), time= 1.6 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R=+685.5  p =  1.2e-160   FAIL !!!!!
  DC6-9x1Bytes-1                    R=+256.6  p =  9.2e-127   FAIL !!!!!
  FPF-14+6/16:(0,14-8)              R= +24.8  p =  6.5e-18    FAIL !
  FPF-14+6/16:(1,14-9)              R= +27.3  p =  1.7e-17    FAIL !
  FPF-14+6/16:(2,14-10)             R= +16.5  p =  2.3e-9   very suspicious
  FPF-14+6/16:(3,14-11)             R= +12.6  p =  2.4e-6   unusual
  FPF-14+6/16:(4,14-11)             R= +14.3  p =  4.5e-7   mildly suspicious
  FPF-14+6/16:all                   R= +44.8  p =  5.1e-29    FAIL !!!
  FPF-14+6/16:cross                 R=+102.7  p =  4.3e-67    FAIL !!!!
  [Low1/8]DC6-9x1Bytes-1            R= +41.0  p =  6.0e-14    FAIL !
  [Low1/8]FPF-14+6/16:cross         R= +11.4  p~=  2.2e-7   very suspicious
  [Low4/32]DC6-9x1Bytes-1           R= +87.7  p =  5.1e-29    FAIL !!!
  [Low4/32]FPF-14+6/16:cross        R= +14.5  p~=  3.7e-9    VERY SUSPICIOUS
  ...and 17 test result(s) without anomalies
```

Figure 8.27: OneRNG cmd0 data file analyzed with PractRand Battery Statistical Test when the device is under the cooling attack

```
rng=RNG_stdin, seed=unknown
length= 512 kilobytes (2^19 bytes), time= 10.9 seconds
  no anomalies in 84 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 12.8 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 14.8 seconds
  no anomalies in 109 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 16.9 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 18.9 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 21.2 seconds
  no anomalies in 151 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 24.6 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 27.6 seconds
  no anomalies in 179 test result(s)
```

Figure 8.28: OneRNG cmd6 and cmd7 data files analyzed with PractRand Statistical Test when the device is under the cooling attack

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/ComScire_A/Heating$ ent -b -c -
t ComScire_A_H150.bin
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,1.000000,0.178443,0.500007,3.141221,0.000019
2,Value,Occurrences,Fraction
3,0,399994026,0.499993
3,1,400005974,0.500007
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/ComScire_A/Heating$ ent -b -c -
t ComScire_A_H200.bin
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,616907064,1.000000,1.206161,0.499978,3.141753,-0.000032
2,Value,Occurrences,Fraction
3,0,308467171,0.500022
3,1,308439893,0.499978
```

Figure 8.29: ComScire data files analyzed with Ent Statistical Test when the device is under the heating attack

```
2023-02-08 09:40:55 ludovica-VirtualBox booltest.booltest_main[4235] INFO Processing finished in 18
.461463451385498 sec
-------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

-------------------------------------------------------------------------------
[+] Results for input: ComScire_A_H150.bin
The best distinguisher found: (x_{9} * x_{70}) + (x_{122} * x_{124})
 - z-score achieved: 5.581804068698074

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.581804068698074 lies in the allowed interval, the randomness hypothesis c
annot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.30: ComScire data files analyzed with BoolTest Statistical Test when the device is under the heating attack (temperature=180°C)

```
2023-02-08 09:48:17 ludovica-VirtualBox booltest.booltest_main[4241] INFO Processing finished in 13
.17374062538147 sec
-------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

-------------------------------------------------------------------------------
[+] Results for input: ComScire_A_H200.bin
The best distinguisher found: (x_{1} * x_{32}) + (x_{137} * x_{193})
 - z-score achieved: 5.593100526508714

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 5.593100526508714 lies in the allowed interval, the randomness hypothesis c
annot be rejected with the alpha 1e-05
 = BoolTest could not find statistically significant non-randomness
```

Figure 8.31: ComScire data files analyzed with BoolTest Statistical Test when the device is under the heating attack (temperature=200°C)

```
rng=RNG_stdin, seed=unknown
length= 512 kilobytes (2^19 bytes), time= 9.2 seconds
  no anomalies in 84 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 megabyte (2^20 bytes), time= 11.1 seconds
  no anomalies in 94 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 megabytes (2^21 bytes), time= 13.0 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+3,13-7,T)                  R= +12.3  p =  2.6e-4   unusual
  ...and 108 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 4 megabytes (2^22 bytes), time= 15.0 seconds
  no anomalies in 124 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 megabytes (2^23 bytes), time= 17.0 seconds
  no anomalies in 135 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 megabytes (2^24 bytes), time= 19.0 seconds
  no anomalies in 151 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 21.2 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 23.8 seconds
  no anomalies in 179 test result(s)
```

Figure 8.32: ComScire data files analyzed with PractRand Statistical Test when the device is under the heating attack

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             ComScire_A/Heating/ComScire_A_H150.bin
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:31.43
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  MultinomialBitsOver             0.9992
 ----------------------------------------------
 All other tests were passed



ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ gcc try1.c -o try1 -ltestu01 -l
probdist -lmylib -lm
ludovica@ludovica-VirtualBox:~/RNG-battery-tests/TestU01/examples$ ./try1

========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             ComScire_A/Heating/ComScire_A_H200.bin
 Number of bits:   100000000
 Number of statistics:  40
 Total CPU time:   00:01:43.46

 All tests were passed
```

Figure 8.33: ComScire data files analyzed with TestU01 Rabbit Battery Statistical Test when the device is under the heating attack

```
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t oneRNGraw-direct-gen-A/Heating/oneRN
Graw_direct_cmd0.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,0.564972,431557441.221863,0.132765,3.990172,0.037497
2,Value,Occurrences,Fraction
3,0,693788169,0.867235
3,1,106211831,0.132765
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t oneRNGraw-direct-gen-A/Heating/oneRN
Graw_direct_cmd6.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,0.697220,311547309.703943,0.187977,3.967902,0.042482
2,Value,Occurrences,Fraction
3,0,649618633,0.812023
3,1,150381367,0.187977
ludovica@ludovica-VirtualBox:~/RNG-battery-tests$ ent -b -c -t oneRNGraw-direct-gen-A/Heating/oneRN
Graw_direct_cmd7.rand
0,File-bits,Entropy,Chi-square,Mean,Monte-Carlo-Pi,Serial-Correlation
1,800000000,0.726070,284019362.637484,0.202080,3.944265,0.065897
2,Value,Occurrences,Fraction
3,0,638335630,0.797920
3,1,161664370,0.202080
```

Figure 8.34: OneRNG data files analyzed with Ent Statistical Test when the device is under the heating attack

```
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw-direct-gen-A/Heating/oneRNGraw_direct_cmd0.rand
The best distinguisher found: (x_{27} * x_{31}) + (x_{137} * x_{141})
 - z-score achieved: 1246.3431177810514

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 1246.3431177810514 lies outside of the interval, the randomness hypothesis
is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.35: OneRNG cmd0 data files analyzed with BoolTest Statistical Test when the device is under the heating attack

```
2023-02-08 10:21:05 ludovica-VirtualBox booltest.booltest_main[4409] Processing finished in 15
.943305253982544 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw-direct-gen-A/Heating/oneRNGraw_direct_cmd6.rand
The best distinguisher found: (x_{160} * x_{164}) + (x_{137} * x_{141})
 - z-score achieved: 1119.0039037130932

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 1119.0039037130932 lies outside of the interval, the randomness hypothesis
is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.36: OneRNG cmd6 data files analyzed with BoolTest Statistical Test when the device is under the heating attack

```
2023-02-08 10:22:01 ludovica-VirtualBox booltest.booltest_main[4416] Processing finished in 16
.11639714241028 sec
--------------------------------------------------------------------------------
BoolTest run with the configuration (256, 2, 2)
 - the block size is 256, i.e., the number of variables in the polynomials. x_{0}, ..., x_{255}
 - the maximal order of the first level terms is 2. Combined by AND.
 - the maximal order of the second level polynomials is 2. Combined by XOR/AND.

--------------------------------------------------------------------------------
[+] Results for input: oneRNGraw-direct-gen-A/Heating/oneRNGraw_direct_cmd7.rand
The best distinguisher found: (x_{121} * x_{125}) + (x_{11} * x_{15})
 - z-score achieved: 1054.3066231018856

BoolTest has reference statistics for used configuration, number of collected samples: 100000
Allowed z-score region in the reference data: [4.775841, 7.609938]
As the achieved z-score 1054.3066231018856 lies outside of the interval, the randomness hypothesis
is REJECTED with alpha 1e-05
 = Data is not random
```

Figure 8.37: OneRNG cmd7 data files analyzed with BoolTest Statistical Test when the device is under the heating attack

```
rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 0.9 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R=+129.4  p =  3.8e-31    FAIL !!!
  DC6-9x1Bytes-1                    R=+220.2  p =  3.5e-94    FAIL !!!!!
  Gap-16:A                          R= +38.5  p =  1.4e-28    FAIL !!!
  Gap-16:B                          R= +51.7  p =  1.4e-33    FAIL !!!
  FPF-14+6/16:all                   R= -21.3  p =1-4.8e-11   VERY SUSPICIOUS
  FPF-14+6/16:cross                 R= +85.6  p =  9.2e-54    FAIL !!!!
  [Low1/8]DC6-9x1Bytes-1            R= +38.0  p =  2.1e-11    FAIL
  [Low1/8]FPF-14+6/16:all           R= -16.7  p~= 1-1e-4    unusual
  [Low4/32]DC6-9x1Bytes-1           R= +30.1  p =  3.3e-9    VERY SUSPICIOUS
  [Low4/32]FPF-14+6/16:all          R= -16.7  p~= 1-1e-4    unusual
  ...and 15 test result(s) without anomalies
```

Figure 8.38: OneRNG cmd0 data files analyzed with PractRand Statistical Test when the device is under the heating attack

```
rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 1.0 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R=+146.0  p =  5.1e-35    FAIL !!!
  DC6-9x1Bytes-1                    R=+206.5  p =  2.5e-88    FAIL !!!!!
  Gap-16:A                          R= +26.9  p =  1.1e-19    FAIL !!
  Gap-16:B                          R= +34.3  p =  3.4e-22    FAIL !!
  FPF-14+6/16:all                   R= -17.3  p =1-6.1e-9    VERY SUSPICIOUS
  FPF-14+6/16:cross                 R= +61.0  p =  1.4e-38    FAIL !!!
  [Low1/8]DC6-9x1Bytes-1            R= +24.0  p =  1.6e-7   very suspicious
  [Low1/8]FPF-14+6/16:all           R= -16.3  p~= 1-1e-4    unusual
  [Low4/32]DC6-9x1Bytes-1           R= +20.1  p =  1.8e-6   very suspicious
  [Low4/32]FPF-14+6/16:all          R= -17.0  p~= 1-9e-5    mildly suspicious
  ...and 15 test result(s) without anomalies
```

Figure 8.39: OneRNG cmd6 data files analyzed with PractRand Statistical Test when the device is under the heating attack

```
rng=RNG_stdin, seed=unknown
length= 2 kilobytes (2^11 bytes), time= 0.3 seconds
  no anomalies in 8 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 kilobytes (2^12 bytes), time= 0.4 seconds
  no anomalies in 12 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 kilobytes (2^13 bytes), time= 0.9 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-9,T)                  R=+115.5  p =  6.7e-28    FAIL !!!
  DC6-9x1Bytes-1                    R=+147.4  p =  3.4e-63    FAIL !!!!
  Gap-16:A                          R=  +9.7  p =  1.8e-6   very suspicious
  Gap-16:B                          R= +13.4  p =  1.8e-8    VERY SUSPICIOUS
  FPF-14+6/16:cross                 R= +33.8  p =  1.1e-21    FAIL !!
  [Low1/8]DC6-9x1Bytes-1            R= +14.8  p =  5.5e-5   suspicious
  [Low1/8]FPF-14+6/16:all           R= -15.7  p~= 1-3e-4    unusual
  [Low4/32]DC6-9x1Bytes-1           R= +25.4  p =  6.7e-8    VERY SUSPICIOUS
  ...and 17 test result(s) without anomalies
```

Figure 8.40: OneRNG cmd7 data files analyzed with PractRand Statistical Test when the device is under the heating attack

9
Conclusion

In this thesis we have analyzed the results obtained when two devices (respectively a TRNG and a QRNG) were subjected to temperature attacks. The devices considered were a TRNG device named OneRNG and a QRNG device named ComScire PQ32MU. These devices have been chosen because they possess a similar mode for generating entropy: in fact both contain a diode and a MOS in reverse bias conditions. First, we generated data from the devices under normal conditions. Then we extracted further data when the devices were under attack: in particular, we performed a cooling attack which envisaged bringing the devices to very low temperatures (-30°C) and a heating attack which instead envisaged bringing both devices to very low temperatures high (80°C, 180°C).

Analyzing the results obtained, it was possible to demonstrate how much the QRNG device is far more resistant and reliable than the TRNG device. Indeed, the ComScire QRNG device was able to resist both attacks, and as soon as it warned (probably through an internal threshold) that it was no longer able to guarantee the reliability of the data produced during the heating attack when we reached a maximum temperature of 200°C, it stopped generating data. On the contrary, this was not visible with regard to the data produced by the OneRNG device: in fact in this case the device continued to produce data even during the heating attack, compromising them. The results obtained show a drastic fall of entropy reaching the value of 0.56.

Each of the data files generated by both devices has been analyzed using four different statistical tests: Ent, PractRand, BoolTest and TestU01, which is the only test that wasn't able to generate result when data were collected with devices under attack.

A special section has been left for the analysis of the performances of the BitReps statistical test, for which it has not been possible to analyze the data files when devices were under attack since, similarly to TestU01, also BitReps is not able to compute the chi-square for not random data: it requires too computational memory. This is the reason why the analysis of this statistical test is performed only analyzing the data files under normal conditions.

For future work, we would like to carry out the same type of experiment this time reaching even lower temperatures than those reached during the cooling attack, so that it is possible to find the minimum value below which the devices stop working in that case as well correctly.

References

[1] J. Mandel, *The Statistical Analysis of Experimental Data*. pp. 15-25, 1914.

[2] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*. pp. 217-228, 2003.

[3] A. B. . J. Wellmann, "Computer simulations then and now: an introduction and historical reassessment," 2019.

[4] Bohan Yang, "Total: Trng on-the-fly testing for attack detection using lightweight hardware," 2016, [Accessed: September 2022].

[5] V. Govindan, "A hardware trojan attack on fpga-based cryptographic key generation: Impact and detection," 2018, [Accessed: September 2022].

[6] S. Ghandali, D. Holcomb, and C. Paar, "Temperature-based hardware trojan for ring-oscillator-based trngs," *arXiv preprint arXiv:1910.00735*, 2019.

[7] K. Yang, D. Fick, M. B. Henry, Y. Lee, D. Blaauw, and D. Sylvester, "16.3 a 23mb/s 23pj/b fully synthesized true-random-number generator in 28nm and 65nm cmos," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 280–281.

[8] Y. Cao, V. Rožić, B. Yang, J. Balasch, and I. Verbauwhede, "Exploring active manipulation attacks on the tero random number generator," in *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2016, pp. 1–4.

[9] M. Varchola and M. Drutarovsky, "New high entropy element for fpga based true random number generators," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2010, pp. 351–365.

[10] NIST. (-) Nist - computer security center resource - random numbers. [Online]. Available: https://csrc.nist.gov/glossary/term/random_number

[11] ——. The concinuous uniform distribution. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section3/eda3662.html

[12] gameludere. (2020) I numeri casuali: Algoritmi di generazione e applicazioni. [Online]. Available: https://www.gameludere.it

[13] W. L. Yuan Cao, "Entropy sources based on silicon chips: True random number generator and physical unclonable function," 2022.

[14] B. Dikshit, "A simple proof of born's rule for statistical interpretation of quantum mechanics," 2017.

[15] J. Cheetham. Onerng website. [Online]. Available: https://onerng.info/

[16] Electrical4U. (2020) Avalanche diode: Working principle & applications. [Online]. Available: https://www.electrical4u.com/avalanche-diode/

[17] Comscire datasheet. [Online]. Available: https://comscire.com/files/datasheet/PQ32MU_datasheet.pdf

[18] Ring oscillator operations & applications. [Online]. Available: https://electricalmag.com/ring-oscillator/

[19] C. Beenakker and C. Schönenberger, "Quantum shot noise," *Physics Today 56, 5, 37; doi: 10.1063/1.1583532*, 2003.

[20] D. Hurley-Smith and J. H. Castro, "Quantum leap and crash: Searching and finding bias in quantum random number generators," 2019.

[21] S. A. Wilber. Entropy analysis and system design for quantum random number generators in cmos integrated circuits. [Online]. Available: https://psigenics.com/files/papers/Pure_Quantum_White_Paper.html

[22] Ent website. [Online]. Available: https://www.fourmilab.ch/random/

[23] Booltest website. [Online]. Available: https://github.com/crocs-muni/booltest

[24] Testu01 website. [Online]. Available: http://simul.iro.umontreal.ca/testu01/tu01.html

[25] P. L'Ecuyer and R. Simard, "Testu01: A software library in ansi c for empirical testing of random number generators," 2013.

[26] Practrand website. [Online]. Available: https://pracrand.sourceforge.net/

[27] Bitreps website. [Online]. Available: https://github.com/jjp31/bitreps-1

[28] Bloom filter. [Online]. Available: https://brilliant.org/wiki/bloom-filter/

[29] S. Infnoise trng. [Online]. Available: https://github.com/waywardgeek/infnoise

Acknowledgments

From September 2022 until December 2022 I was given the opportunity to carry out a mobility period at the University of Kent (Kent, UK), supervised by professors Mauro Conti and Julio Hernandez Castro. Furthermore, I had the opportunity to collaborate with three PhD students, Maryam Ehsanpour, Jamie Pont and Calvin Brierley: I want to thank each of you for making this experience possible.

A big thank you also goes to the CY4GATE company and in particular to the CY4WOMEN campaign for having contributed financially to this period of mobility.