Chapter 1

Randomness Extractor

A Quantum Random Number Generator can produce random numbers with certified randomness, in practice, however, quantum signals (the source of true randomness) are inevitably mixed with classical noises. An adversary (Eve) can, in principle, control the classical noise and gain partial information about the raw random numbers. Therefore, it is necessary to apply a postprocessing procedure to filter the true randomness of which Eve has almost no information. This procedure is called randomness extraction, realized by using randomness extractors [1,2]. Hence the aim of randomness extractors is to extract (almost) perfect randomness, i.e. uniformly distributed randomness, from the raw data generated from a practical QNRG with the help of a random seed, not necessarily uniform [3]. The main parameter of a randomness extractor is the min-entropy of raw data based on the concept of guessing probability. Having specified a randomness quantifier, we need to define what we mean by an "almost uniform" distribution over the output $Z = \mathrm{Ext}(X, Y)$, where X is the source and Y the seed. Firstly, we have to introduce the following definition:

> **Definition 1.0.1** (ϵ**-close**)**.** *Two probability distributions X and Y over the same set T are ϵ-close if the statistical distance bewteen them is bounded by ϵ,*
>
> $$||X - Y|| := \frac{1}{2} \sum_{v \in T} |P(X = v) - P(Y = v)| \leq \epsilon. \tag{1.1}$$

An extractor can be defined as follows:

> **Definition 1.0.2** ((k, ϵ)**-extractor**)**.** *A function* $\mathrm{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ *is a (k, ϵ)-extractor if for every distribution X over $\{0,1\}^n$ with $H_\infty(X) \geq k$,* $\mathrm{Ext}(X, Y)$ *is ϵ-close to uniform, where Y is distributed uniformly over $\{0,1\}^d$.*

However, we are in presence of an adversary, thus we would like to make sure that someone who has some side information cannot do better than with a random guess, that is, the probability that he guesses correctly should be close to $\frac{1}{2^m}$ if the output is a bit string of length m. We suppose that the inputs are bit strings of length n with min-entropy of at least k, which are processed using a randomly distributed

seed of length d. The output are bit strings of length m, and the distribution of outcomes should be ϵ-*close* to uniform distribution and independent from the side information. The length of the output of the extractor is m and we want m as close to k as possible, to extract most of the entropy. The value $k - m$ is therefore called the *entropy loss*. We also would like to have the seed's size d as small as possible, to minimise the amount of additional randomness.
Previously, some simple methods have been used for randomness extraction, for example a bitwise XOR operation between two halfs bit strings of a raw data string [4]; or nonuniversal hashing functions [5]. However, the randomness generated through these methods is not theoretically provable to be random. Recently, an important randomness extractor, called Trevisan's extractor [6], has received a considerable theoretical interest because it is safe against quantum opponents [3]. The seed length of Trevisan's extractor is polylogarithmic in the length of the input, outperforming randomness extractors based on universal hashing, which are now more often used in quantum cryptography, but require a seed size that scales linearly with the length of the raw input. Moreover, it is also proven that Trevisan's extractor is a strong extractor [7], i.e. the seed is independent of the final output, thus the randomness in the seed is not consumed in the process (compared to weak extractors) and can be reused as part of the result. Two implementation of this extractor were made by Ma et al. [8] and Mauerer et al. [9].

1.1 Trevisan's Extractor

The major contribution of Trevisan derives from the intuition that a particular class of error correction codes (ECC), called *list-decodable codes* [10], can be interpreted as extractors. An error correcting code is a deterministic function that map from shorter into longer bit strings to make them more robust against possible errors that can act on the encoded data.

> **Definition 1.1.1** (**List-decodable code**). *For $x, y \in \{0,1\}^n$, let $\delta(x,y)$ denote the relative Hamming distance of x and y, that is $\delta(x,y) = \frac{|\{i : x_i \neq y_i\}|}{n}$ A function $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ is an (l,ϵ)-list-decodable code if for every $z \in \{0,1\}^{\bar{n}}$, $|\{x : \delta(C(x), z) \leq \frac{1}{2} - \epsilon\}| \leq l$.*

This definition states that if we encode a string $x \in \{0,1\}^n$ by $C(x)$ and then send $C(x)$ on a "noisy channel" which is allowed to flip $1/2 - \epsilon$ of the indices of $C(x)$ to obtain a string z, then, who receives z knows that the original message x can correspond to a maximum of l different messages. The first consideration of Trevisan was that if we randomly choose a position of an ECC's output string, the corresponding extracted bit is uniformly distributed, provided that the initial distribution has enough min-entropy [6]. Formally, if $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ is an (l,ϵ)-list-decodable code then the function

$$\begin{aligned} C' : \{0,1\}^n \times [\bar{n}] &\to \{0,1\} \\ (x,y) &\mapsto C(x)_y \end{aligned} \tag{1.2}$$

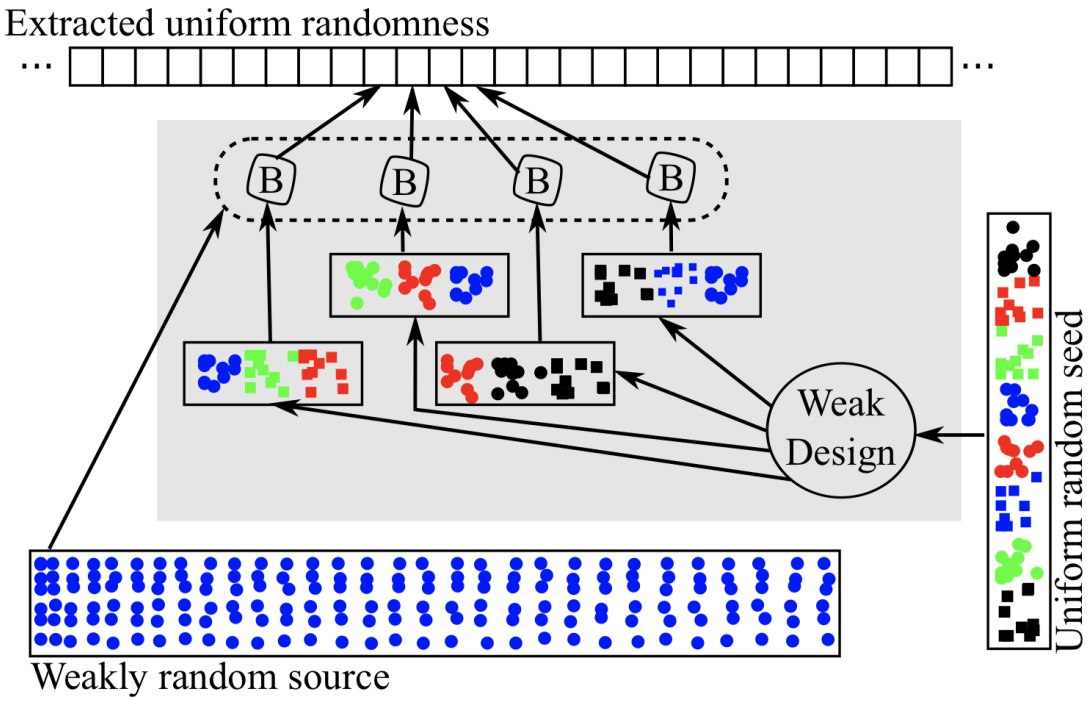

Figure 1.1. Extraction process. The image represents the extraction process devised by Trevisan: in particular we have a uniform seed of length d which is subdivided into m smaller packages (sub-seed) with a certain overlap between them through weak design. Each sub-seed is fed into a one-bit extractor along with the weak source with min-entropy of k. For each one-bit extractor a bit is extracted, so in total the output string will have $m < k$ bits, according to the algorithm used. Figure from [9].

is a one-bit extractor, where $C(x)_y$ is the yth bit of $C(x)$.
Trevisan's second observation states that we can concatenate the outputs of the one-bit extractor so that the total output is still uniformly distributed. Furthermore Trevisan claims that we do not need to choose a completely new random seed for every application of the one-bit extractor. The principal idea is that the initial seed is divided into sets of random bits with small overlap. If the overlap is not too large, there are not too many correlations using the elements of each set into the one-bit extractors and then concatenating the total output bits. In order to divide the initial seed bits into smaller sets we use an algorithm called *weak design* (WD).
Therefore the weak design spreads the initial seed across different smaller sets with a certain overlap. The overall length can be bigger than the seed length. Each set is sent to a one-bit extractor, which extracts, for each packet, a single random bit from the random source. These bits are finally concatenated into a string to generate the extracted random string. The complete process is shown in fig. 1.1. Two main component are required to construct Trevisan's extractor: a one bit extractor and a weak design. Therefore Trevisan's construction is not really a single algorithm, but a prescription to put together different one-bit extractors and weak designs to generate a quantum strong extractor.

In our case we realized the one-bit extractor using an error correcting code, which is constructed by concatenating a Reed-Solomon code with an Hadamard code (see section 1.3), while for the weak design, a refined version of the original construction from Nisan and Wigderson is used [11] (see section 1.2).
As we have seen, Trevisan's algorithm consists in applying several times the same one-bit extractor to the input string, using, for each run, different seeds which are weakly correlated. The seeds are obtained as substrings from a longer seed $y \in \{0,1\}^d$. More formally, let $\{S_i\}$ be a family of sets such that for all i, $|S_i| = t$ and $S_i \subset [d] = \{1, ..., d\}$. Then y_{S_i}, the string formed by the bits of y at the positions given by the elements $j \in S_i$, is a string of lenght t. For a given one-bit extractor $C' : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}$ and such a family $\{S_i\}$ with $1 \leq i \leq m$, Trevisan's extractor is defined as the concatenation of the output bits of C with seed y_{S_i} used, that is:

$$\mathrm{Ext}(x, y) := C'(x, y_{S_1}) \cdots C'(x, y_{S_m}). \tag{1.3}$$

where $y_{S_i} \in \{0,1\}^t$ can be interpreted to be an integer in $[\bar{n}]$.
The performance of the extractor depends on the performance of the one-bit extractor, but also on the overlap of the seeds used for each run of the one-bit extractor. Intuitively, we can extract more randomness from the source if the cardinality of the intersections of the sets $\{S_i\}$ is small, but this implies a larger seed. Furthermore if ϵ is the error of the one bit extractor (according to def. 1.0.1), it is also the error per bit for Trevisan's construction.

Definition 1.1.2 (**Weak Design [7]**). *A family of sets $S_1, \ldots S_m \subset [d]$ is a **weak (m, t, r, d)-design** if*

1. $|S_i| = t \ \forall i$

2. $\sum_{j=1}^{i-1} 2^{|S_j \cap S_j|} \leq rm \ \forall i$

*where the parameter r is the so called **overlap** of the weak design and depends on the algorithm used.*

Summing up:

- n is the input length;
- m is output length and it is also the number of substrings in which the seed is divided;
- d represents the seed length of Trevisan's extractor;
- t the length of the substrings fed into the one bit extractors;
- r is the overlap of the weak design;
- k is the min-entropy required in the source, which is often expressed as $k = \alpha n$, where α is the min-entropy per bit (usually $\alpha < 1$);
- finally ϵ is the error per bit for the final construction.

We take as input the source length n, the min entropy k and the error per bit ϵ (arbitrarily set by the user), and then we calculate the seed length d and output length $m < k$, in fact, with our algorithm, we are not able to extract the maximum number k of extractable bits.

1.2 Weak Design

1.2.1 Standard Weak Design

For the weak design we use a refined version of the original construction from Nisan and Wigderson [11]. Mauerer et al. [9] proved that this construction is a weak (m, t, r, d)-design for any prime power t, any m and that $r = 2e$ and $d = t^2$. We will refer to this construction as standard weak design (SWD). Obviously a larger overlap produces a greater loss of entropy, hence we adopt the idea proposed by Mauerer [9] to construct a new design with $r = 1$, starting from an iterative construction of the basic design from Ma and Tan [12]. In general, the construction of a weak design is based on the use of polynomials over a finite field $\mathrm{GF}(t)$.
We can index every set S_p by one such polynomial p: $\mathrm{GF}(t) \to \mathrm{GF}(t)$. To construct a weak (m, t, r, d)-design we need to have m sets, and thus m such polynomials, taken in increasing order of their coefficients. In general, we define the nth polynomial as

$$p(x) = \sum_{i=0}^{c} \alpha_i x^i \quad \text{with} \quad \alpha_i = \lfloor (n-1) \rfloor / t^i \bmod t \quad \text{and} \quad c = \left\lceil \frac{\log m}{\log t} - 1 \right\rceil. \tag{1.4}$$

Each element of S_p is a pair of the form $(x, p(x))$ $\forall x \in \mathrm{GF}(t)$, therefore

$$S_p := \{(x, p(x)) : x \in \mathrm{GF}(t)\}. \tag{1.5}$$

Hence, each set consists of $|S_p| = t$ elements. If we fix $d = t^2$ then, considering the pair $(x, p(x))$ as a two-digit number (x and $p(x)$) and converting it to base t, we obtain that the set S_p contain only elements which are numbers less then d.

1.2.2 Block Weak Design

In order to reduce to overlap r, Mauerer et al. [9] noticed that any weak design is actually a binary $(m \times d)$-matrix W, where the elements $w_{i,j} = 1$ if $j \in S_i$. A weak design with $r = 1$ can be made using repeatedly the previous weak design construction with different values m_j (but the same t), achieving different designs $W_{B,0}, \ldots, W_{B,l}$. Thus we can build a new design W by putting these designs in its diagonal, that is:

$$W = \begin{pmatrix} W_{B,0} & & \\ & \ddots & \\ & & W_{B,l} \end{pmatrix}$$

where l is defined as

$$l := \max\left\{1, \left\lceil \frac{\log(m - r') - \log(t - r')}{\log(r') - \log(r' - 1)} \right\rceil \right\} \tag{1.6}$$

letting $r' = 2e$ be the parameter from the basic contruction and m and t be fixed to the values required by the one bit extractor . Each design $W_{B,i}$ is constructed with m_i sets defined as follows:

$$n_i := \left(1 - \frac{1}{r'}\right)^i \left(\frac{m}{r'} - 1\right) \qquad 0 \leq i \leq l-1, \tag{1.7}$$

$$m_i := \left\lceil \sum_{j=0}^{i} n_j \right\rceil - \sum_{j=0}^{i-1} m_j \qquad 0 \leq i \leq l-1, \tag{1.8}$$

$$m_l := m - \sum_{j=0}^{l-1} m_j. \tag{1.9}$$

Thus weak design W has $d = (l+1)t^2$. Mauerer et al. [9] proved that this construction, called *block weak design* (BWD), has $r = 1$.

1.3 One-bit Extractor

The construction of our one-bit extractor makes use of hash functions. An hash function is an algorithm that maps data of arbitrary length (message) in a fixed-length binary string called *hash value*. This function is designed to be unidirectional (one-way), i.e. a function difficult to reverse: the only way to recreate the input data from the output of an ideal hash function is to try a brute-force search for possible input to see if there is a match.
The one-bit extractor used is the concatenation of two hash functions, Reed-Solomon code and Hadamard code, which combined become a list-decodable code, as showed by Guruswami et al. [13], and thus a good one-bit extractor.
This construction uses a seed of length $2l := t$.
Regarding the first code we divide the input string $x \in \{0,1\}^n$ in blocks $x = (x_1, \ldots, x_s)$, each of length l and if necessary we pad the last string x_s with 0s. Then we treat each block as an element of a finite field, i.e. $x_i \in \mathrm{GF}(2^l)$, and compute the polynomial:

$$p_\alpha(x) = \sum_{i=1}^{s} x_i \alpha^{s-1} \tag{1.10}$$

where $\alpha \in \mathrm{GF}(2^l)$ is the first half of the seed.
Then, we use it with another hash function, the *Hadamard code*. This hash function calculate the parity of the bitwise product of $p_\alpha(x)$ and the second half of the seed, $\beta \in \{0,1\}^l$. Hence we obtain

$$z = \bigoplus_{i=1}^{l} \beta_i p_\alpha(x)_i. \tag{1.11}$$

Mauerer et al. proved that the seed length of this one bit extractor is [9]:

$$t = 2l = 2\lceil \log n + 2\log 2/\epsilon \rceil, \tag{1.12}$$

where the seed length of the complete construction is $d = t^2$ and the randomness required is $k = rm + 4\log\frac{1}{\epsilon} + 6$ [9]. Thus, for a given min-entropy k and a fixed error per bit ϵ the number of bits that we can extract is

$$m = \lfloor (k - 4\log_2(1/\epsilon) - 6)/r \rfloor). \tag{1.13}$$

Below are shown the graphs depicting the parameters behaviour of our extractor.

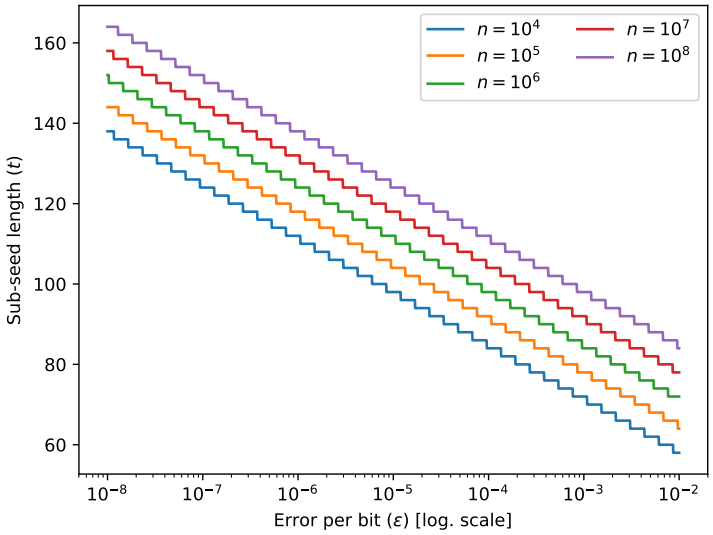

Figure 1.2. Length of the single set S of the weak design depending on the error per bit. It can be noted that in semi-log scale, the sub-seed length t is a step function and decreases linearly as the error per bit ϵ increases. Furthermore, the greater the input n, the greater the length of the single sub-seed created by the weak design.

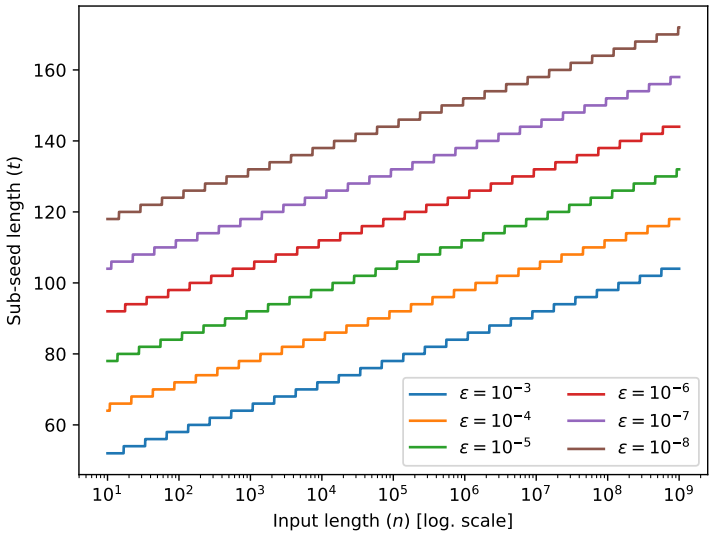

Figure 1.3. Plot of the length of the single set S of the weak design vs the input length of the source. In this figure is represented how the sub-seed length t varies as a function of the input length n, plotted for different error per bit ϵ parameters. We can see that in semi-log scale the sub-seed length is a step function and increases linearly with the input. Furthermore, the greater the error per bit, the smaller the length of the single sub-seed created by the weak design.

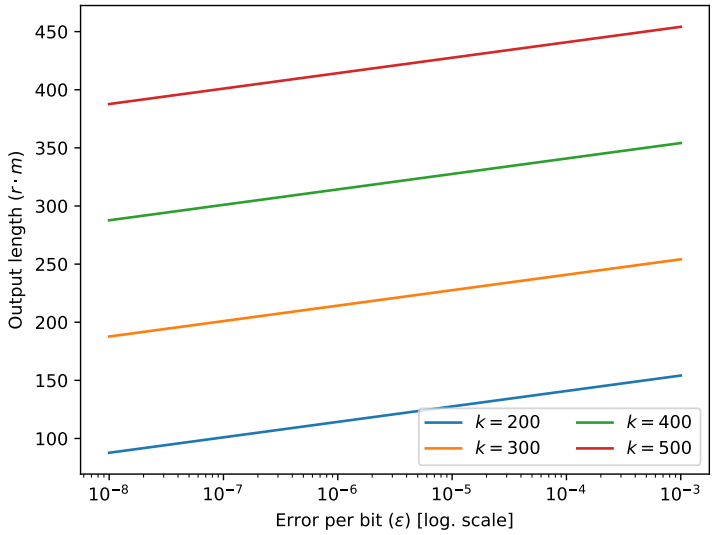

Figure 1.4. Ratio between the output length of the extractor and the error per bit parameter. This figure shows linear dependence in semi-log scale of the output length m (multiplied by a constant factor r) as a function of the error per bit ϵ. Output length increases with the error and the greater the min-entropy k, the greater the output length.

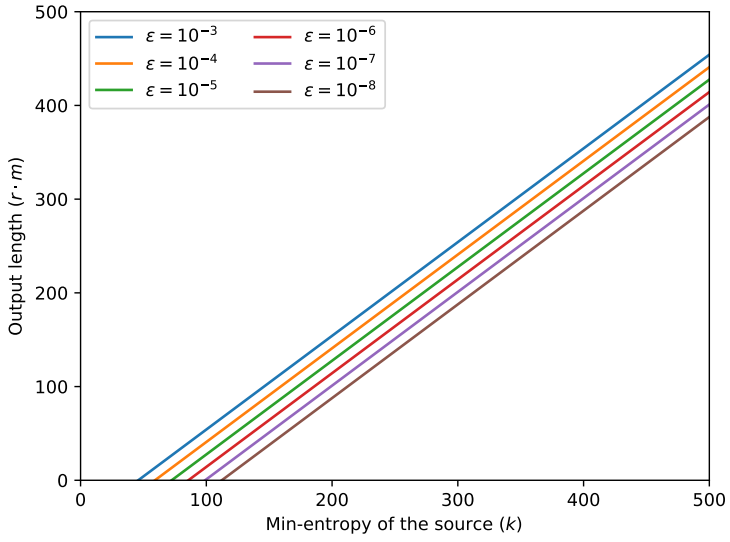

Figure 1.5. Output length vs the min-entropy of the source. It can be noted that the output length m (multiplied by a constant factor r) is a linear growing monotone function of the min-entropy of the source k and it also increases with the error per bit ϵ.

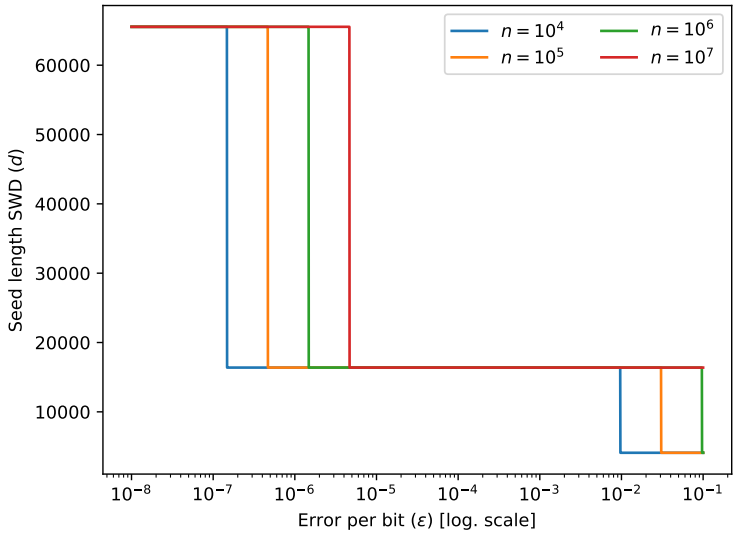

Figure 1.6. Dependency of seed length from the error per bit. Here different graphs of the seed length d as a function of the error per bit ϵ are plotted, for different input source length in semi-log scale. These functions are step functions and, with the same error, the seed increases with the size of the input n.

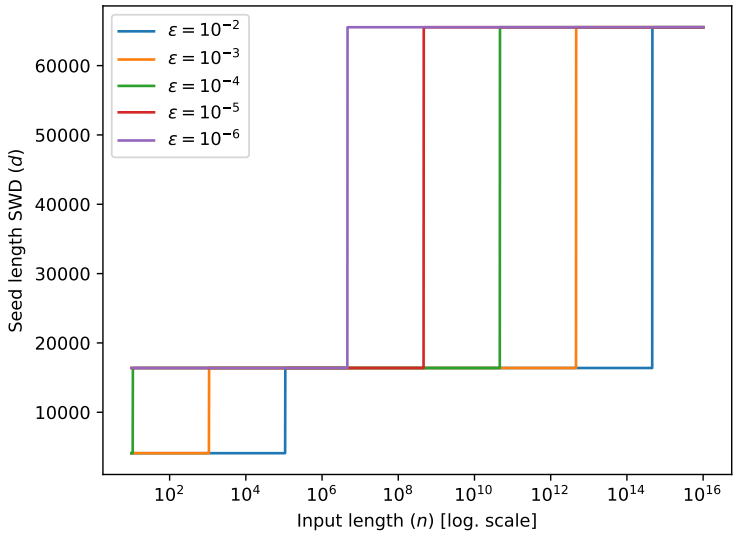

Figure 1.7. Comparison between seed length and the input length of the source. In this figure is represented the seed length d vs the input length n in semi-log scale, plotted for different error per bit ϵ parameters. The seed is a step function of the input length, it increases with n and, with the same input length, the seed length is greater for lower errors.

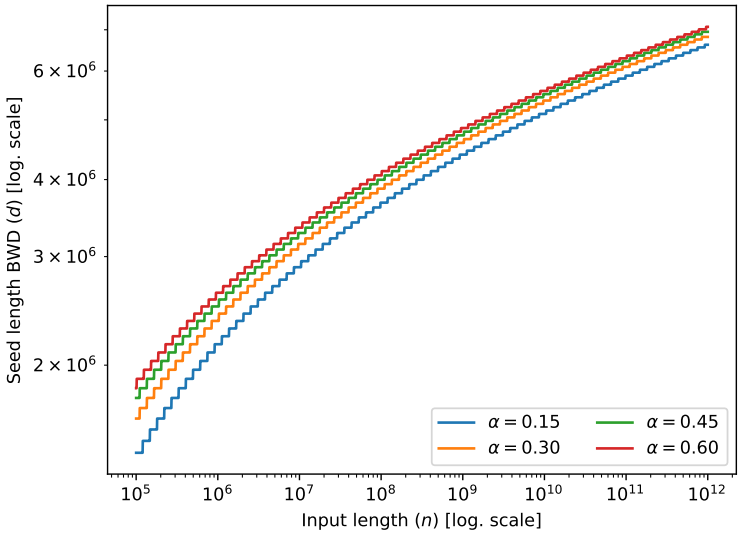

Figure 1.8. Relation between the seed length of block weak design and the input length of the source. The seed length d as a function of the input length n is plotted for different values of min-entropy per bit α and the error per bit ϵ parameter is fixed at $\epsilon = 10^{-7}$. d is a monotone increasing function of n and it increases also with α. Both the axes are in logarithmic scale.

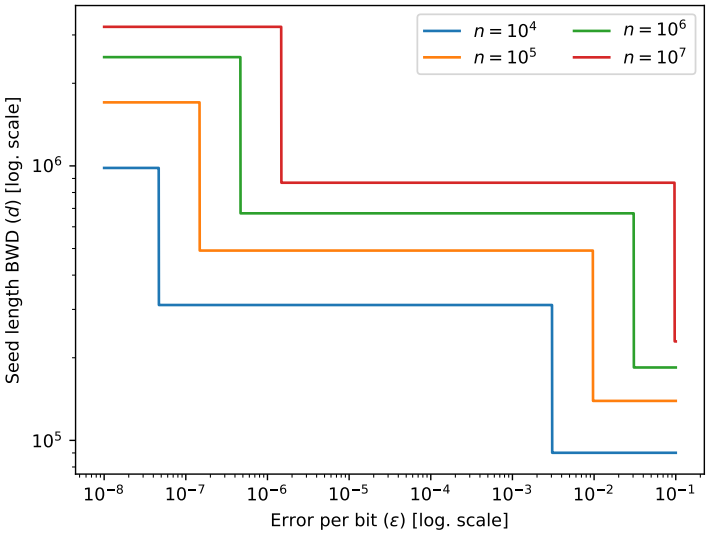

Figure 1.9. Seed length of block weak design depending on the error per bit. For different input length of the source, the seed length d as function of the error per bit ϵ is plotted. d is a descending step function of ϵ, but it increases with n. The min-entropy is fixed at $\alpha = 0.4$ and both the axes are in logarithmic scale.

1.4 Algorithm Implementation

Now we are going to present the pseudo code of a modified version from Mauerer *et al.* [9], the full source code is available at `https://github.com/michelemancusi/libtrevisan`.

Weak Designs:

a. Construction of Hartman and Raz (standard weak design)

Determining d from t is not so simple, we have to pay attention to some constraints: the $GF(2^x)$, on which the weak design is based, only works with values of t that are powers of 2, thus we need to choose larger values of t than requested from one-bit extractor (and this implies a bigger initial seed d), because the one-bit extractor works with arbitrary values of t. Thus we need to increase t to the nearest power of 2. Therefore we have to distinguish between t, that is the value relative to the weak design used to carry out calculations in $GF(t)$, and t_{req}, which is the value required by the bit extractor. This implies necessarily that $t \geq t_{req}$.

1: **procedure** SWDCOMPUTE$S_i(i, m, t, t_{req})$
2: $c = \lceil \log_2(m)/\log_2(t_{req}) - 1 \rceil$
3: $\text{mask} = (1 \ll \log_2(t)) - 1$
4: **for** $j \leq c$ **do**
5: $\alpha[j] = (i \wedge (\text{mask} \ll j \cdot \log_2(t))) \gg j \cdot \log_2(t)$
6: **end for**
7: **for** $a < t_{req}$ **do**
8: $b = 0$
9: $Sa = 0$
10: **for** $k \leq c$ **do**
11: $b \leftarrow b + \alpha[k] \cdot a^k$ ▷ All this operations are computed over $GF(t)$
12: **end for**
13: $Sa = Sa \oplus b$
14: $Sa = Sa \oplus (a \ll \log_2(t))$
15: $S[a] = Sa$
16: **end for**
17: **return** S
18: **end procedure**

Finite fields of order 2^m are called binary fields or characteristic-two finite fields. In finite fields $GF(2^m)$ all elements of the field are interpreted as polynomials over the binary field, i.e. polynomials whose coefficients are either 0 or 1. In this field there are 2^m polynomials with degree no more than $m - 1$, thus the elements of this field can be viewed as m-bit strings. Each bit in the bit string corresponds to the coefficient of the polynomial taken at the same position.
Additions and subtractions in a finite field are performed by adding or subtracting two of the polynomials of this field together, and then reducing the result modulo 2, while a multiplication can be done as a multiplication modulo an irreducible polynomial used to define the field. Multiplication of polynomials over the binary

field can be implemented as simple bit-shift and XOR. The procedure is the following: take two polynomials written as bit strings; perform the classical multiplication considering the two bit strings as integer numbers; if the degree d of the resulting polynomial is greater than $m-1$, shift the irreducible polynomial l bit left, where $l = d - m$ and then perform bitwise XOR. Repeat the previous point until you get a polynomial of degree $m-1$.

b. Block Weak Design

The block weak design is founded on a basic design whose matrix representation is used several times to obtain the total weak design: once the matrix representation of the basic design has been calculated, to build the complete design, we have to place sub-matrices of the basic design matrix on the diagonal of a larger matrix. If we consider the basic design as a vector of indices instead of a matrix, it is possible to compute the elements of $W_{B,j}$ from the basic row $W_{B,0}$ by adding $j \cdot t^2$ to all values of the set S corresponding to the matrix row.

1: **procedure** BWDCOMPUTE$S_i(ic, i, m, t, t_{req})$
2: $r' = 2e$
3: $l = \max\left\{1, \left\lceil \frac{\log(m-r') - \log(t-r')}{\log(r') - \log(r'-1)} \right\rceil \right\}$
4: $j = i \bmod l$
5: $k_1 = i/l$;
6: $n_0 = (m/r' - 1)$
7: $m_0 = \lceil n0 \rceil$
8: **if** $k_1 \neq ic$ **then**
9: $ic = k_1$
10: SWDcompute$S_i(ic, m0, t, t_{req})$
11: **for** $h < t_{req}$ **do**
12: $S_c[h] = S[h]$
13: **end for**
14: **else**
15: **for** $h < t_{req}$ **do**
16: $S[h] = S_c[h] + j \cdot t^2$
17: **end for**
18: **end if**
19: **return** S
20: **end procedure**

1-Bit extractor

The algorithm used to perform the one-bit extractor through hash functions is based on a concatenation of a Reed-Solomon and a Hadamard code as follows:

```
1: procedure 1-Bit extractor(seed, source, n, ε)
2:     l = ⌈log_2(n) + 2 log_2(2/ε)⌉
3:     s = ⌈n/l⌉
4:     b = 0
5:     for i < n do
6:         totalsource[i] = source[i]
7:     end for
8:     for i = n; i < s · l do
9:         totalsource[i] = 0
10:    end for
11:    for i < l do
12:        r[i] = 0
13:    end for
14:    for i < s do
15:        for j < l do
16:            c[i][j] = totalsource[i · l + j]
17:        end for
18:    end for
19:    for i < l do                                   ▷ Reed-Solomon step
20:        α[i] = seed[i]
21:    end for
22:    for j = 1; j ≤ s do
23:        a = c[j − 1] · α^(j−1)        ▷ All this operations are computed over GF(2^l)
24:        for i < l do
25:            r[i] = r[i] ⊕ a[i]
26:        end for
27:    end for
28:    for i < l do                                   ▷ Hadamard step
29:        b = b ⊕ (seed[i + l] ∧ r[i])
30:    end for
31:    return b
32: end procedure
```

Trevisan's extractor

The Trevisan algorithm does not depend on the type of weak design and bit extractor used, only the deduced parameters depend on the specific characteristics of the parts.

1: **procedure** TREVISAN(**WeakDesign, 1-Bit extractor**, seed, source, n, α, ϵ)
2: $k = \alpha \cdot n$
3: $m = \lfloor (k - 4\log_2(1/\epsilon) - 6)/r \rfloor$ ▷ where $r = 2e$ or $r = 1$ depending on the weak design you use.
4: $t_{req} = 2\lceil \log_2(n) + 2\log_2(2/\epsilon) \rceil$
5: $t = 2^{\lceil \log_2(t_{req}) \rceil}$
6: **for** $i < m$ **do**
7: $S =$WDcompute$S_i(i, m, t, t_{req})$ ▷ where here you can use standard weak design or block weak design
8: $b = 0$
9: **for** $j < t_{req}$ **do**
10: $b[j] = \text{seed}[S[i][j]]$
11: **end for**
12: $\rho[i] =$1-Bit extractor$(b, \text{source}, n, \epsilon)$
13: **end for**
14: **return** ρ
15: **end procedure**

Bibliography

[1] Noam Nisan and Amnon Ta-Shma. Extracting randomness: A survey and new constructions. *J. Comput. Syst. Sci.*, 58(1):148–173, 1999.

[2] Ronen Shaltiel. An introduction to randomness extractors. In *International Colloquium on Automata, Languages, and Programming*, pages 21–41. Springer, 2011.

[3] Anindya De, Christopher Portmann, Thomas Vidick, and Renato Renner. Trevisan's extractor in the presence of quantum side information. *SIAM Journal on Computing*, 41(4):915–940, 2012.

[4] Robert Churchhouse and RF Churchhouse. *Codes and ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press, 2002.

[5] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[6] Luca Trevisan. Extractors and pseudorandom generators. *Journal of the ACM*, 48(4):860–879, 2001.

[7] Ran Raz, Omer Reingold, and Salil Vadhan. Extracting all the randomness and reducing the error in trevisan's extractors. *Journal of Computer and System Sciences*, 65(1):97–128, 2002.

[8] Xiongfeng Ma, Feihu Xu, He Xu, Xiaoqing Tan, Bing Qi, and Hoi-Kwong Lo. Postprocessing for quantum random-number generators: Entropy evaluation and randomness extraction. *Physical Review A*, 87(6):062327, 2013.

[9] Wolfgang Mauerer, Christopher Portmann, and Volkher B Scholz. A modular framework for randomness extraction based on trevisan's construction. *arXiv preprint arXiv:1212.0520*, 2012.

[10] Salil Vadhan. The unified theory of pseudorandomness. In *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010) (In 4 Volumes) Vol. I: Plenary Lectures and Ceremonies Vols. II–IV: Invited Lectures*, pages 2723–2745. World Scientific, 2010.

[11] Noam Nisan and Avi Wigderson. Hardness vs randomness. *Journal of computer and System Sciences*, 49(2):149–167, 1994.

[12] Xiongfeng Ma, Zhen Zhang, and Xiaoqing Tan. Explicit combinatorial design. *arXiv preprint arXiv:1109.6147*, 2011.

[13] Venkatesan Guruswami, Johan Hastad, Madhu Sudan, and David Zuckerman. Combinatorial bounds for list decoding. *IEEE Transactions on Information Theory*, 48(5):1021–1034, 2002.