HAL
archives-ouvertes.fr

A Comparison of Post-Processing Techniques for Biased Random Number Generators

Siew-Hwee Kwok, Yen-Ling Ee, Guanhan Chew, Kanghong Zheng, Khoongming Khoo, Chik-How Tan

▶ **To cite this version:**

Siew-Hwee Kwok, Yen-Ling Ee, Guanhan Chew, Kanghong Zheng, Khoongming Khoo, et al.. A Comparison of Post-Processing Techniques for Biased Random Number Generators. Claudio A. Ardagna; Jianying Zhou. 5th Workshop on Information Security Theory and Practices (WISTP), Jun 2011, Heraklion, Crete, Greece. Springer, Lecture Notes in Computer Science, LNCS-6633, pp.175-190, 2011, Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication. <10.1007/978-3-642-21040-2_12>. <hal-01573305>

HAL Id: hal-01573305
https://hal.inria.fr/hal-01573305

Submitted on 9 Aug 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

A Comparison of Post-Processing Techniques for Biased Random Number Generators

Siew-Hwee Kwok[1], Yen-Ling Ee[1], Guanhan Chew[1], Kanghong Zheng[1], Khoongming Khoo[1] and Chik-How Tan[2]

[1]DSO National Laboratories, 20 Science Park Drive, S118230, Singapore.
{ksiewhwe,eyenling,cguanhan,zkanghon,kkhoongm}@dso.org.sg
[2]Temasek Laboratories, National University of Singapore.
tsltch@nus.edu.sg

Abstract. In this paper, we study and compare two popular methods for post-processing random number generators: linear and Von Neumann compression. We show that linear compression can achieve much better throughput than Von Neumann compression, while achieving practically good level of security. We also introduce a concept known as the adversary bias which measures how accurately an adversary can guess the output of a random number generator, e.g. through a trapdoor or a bad RNG design. Then we prove that linear compression performs much better than Von Neumann compression when correcting adversary bias. Finally, we discuss on good ways to implement this linear compression in hardware and give a field-programmable gate array (FPGA) implementation to provide resource utilization estimates.

Keywords. bias, linear correcting codes, entropy, random number generators, post-processing.

1 Introduction

Hardware-based random number generators (HRNGs) are sometimes preferred over algorithm-based bit generators. The randomness in the raw bitstream generated by HRNGs depend on the highly unpredictable nature of certain physical processes and are therefore less prone to the risks of cryptanalytic attacks, which are more applicable on deterministic bit generators. However, the raw output of HRNGs tends to be slightly biased and may even deteriorate over time.

To address this problem, HRNG implementations typically add an additional post-processing step to ameliorate symptoms of non-randomness in the raw bitstream. Basically, a compression function is applied to the raw bitstream before it is output at the user's end. If the raw bitstream starts out with an unusually high bias, the post-processing step would transform the bitstream such that the bias becomes more acceptable. Sometimes this may come at the expense of throughput, for example, the processed bitstream of [4] is half as long as the raw bitstream.

Common techniques used in the post-processing step include hashing or block-wise XOR-ing. A well-known method that makes use of the Von Neumann corrector [3] is sometimes used. Each of these methods has its pros and

cons. Another technique, proposed by several authors [4, 6, 9] recently, uses linear compression functions based on good linear codes. Since there is a large pool of linear codes to choose from, it becomes possible to trade-off different aspects of a HRNG's performance, unlike in the other methods. For example, we show in Section 3 that if a RNG has random bias 0.001, using Von Neumann compression will produce perfect correction with bias= 0 but throughput equal to 25% of its original. However, if we use a $[255, 191, 17]$ BCH code, we could get a good bias of 2^{-153} and three times the throughput at 75% of the original transmission speed.

In Section 4, we introduce a concept called the adversary bias, which measures how accurately an adversary can guess the output of a RNG. This may occur, for example, if a user buys a RNG from a dishonest vendor who installed a trapdoor; or it may arise from an inherent weakness/bad design of a RNG. We show that linear compression can lower the adversary bias by much more than Von Neumann compression. In [6, 9], the authors suggested using BCH codes with parameters $[255, 21, 111]$ and $[256, 16, 113]$ for linear compression, which do not seem to offer any advantage over Von Neumann compression in terms of throughput and correction of random bias. However, we show that these BCH codes are many times more effective than Von Neuman compression for correcting the adversary bias.

In addition, we look at two explicit constructions for implementing linear corrector functions based on BCH codes. One is based on multiplication by the generator polynomial, while the other is based on taking the remainder after division by the parity check polynomial. We compare the two methods and show when one is more advantageous over the other for different parameters. Finally, we implement this post-processing function in field-programmable gate array (FPGA) hardware circuitry to provide some resource utilization figures.

2 Known Techniques for De-Biasing

2.1 Compression with Cryptographic Hash

A cryptographic hash function is a deterministic algorithm that takes in an arbitrary block of data as input and returns a string of fixed size as output. When the size of the data input is larger than the stipulated size of the output, a compression is done on the data block by the hash function. The fundamental requirements of a cryptographic hash function are one-wayness, pre-image resistance, second pre-image resistance and collision-resistance. Examples of cryptographic hash functions include the SHA family of hash functions [1].

In [2], it is stated that if the data input has high entropy, the cryptographic hash output on this data will be close to a uniform distribution, thus de-biasing the input. However, it is hard to quantify how good the output bias is with respect to the input bias. Moreover, as hash functions are non-linear functions, there are hardware limitations and they may not be efficient to implement.

It is common to use hash functions for post-processing RNG output. However, because it is hard to quantify their strength and compare with other de-biasing methods, we shall leave hash functions out of our discussion in this paper.

2.2 Compression Using the Von Neumann Corrector

The Von Neumann corrector [3] is a well known method for post-processing a biased random stream. It is a simple method that produces perfectly unbiased outputs. Suppose an input stream has independent but biased bits. The corrector processes the stream of bits as a stream of non-overlapping pairs of successive bits and generates outputs as follows:

(1) If the input is "00" or "11", the input is discarded (no output),
(2) If the input is "01" or "10", output the first bit only.

Suppose the input bits have bias e , this means that for an input bit x,

$$Pr(x = 0) = \frac{1}{2} + e \text{ and } Pr(x = 1) = \frac{1}{2} - e. \tag{1}$$

Then for a given output bit y,

$$\begin{aligned} Pr(y = 1) &= Pr(y = 1| \text{ there is an output }) \\ &= \frac{Pr(\text{“10”})}{Pr(\text{“01”or“10”})} \\ &= \frac{(\frac{1}{2} - e)(\frac{1}{2} + e)}{(\frac{1}{2} + e)(\frac{1}{2} - e) + (\frac{1}{2} - e)(\frac{1}{2} + e)} \\ &= \frac{\frac{1}{4} - e^2}{2(\frac{1}{4} - e^2)} \\ &= \frac{1}{2}. \end{aligned}$$

Thus the Von Neumann corrector output bits with zero bias.

However, the rate of such a corrector is fairly slow. The rate a desirable pair (i.e. "01" and "10") occurs is $2(\frac{1}{2} + e)(\frac{1}{2} - e)$ which is $2(\frac{1}{4} - e^2)$. However, each pair gives an output half its length. Hence the rate of the corrector is given by $\frac{1}{4} - e^2$. Thus the rate of the Von Neumann corrector is at best $\frac{1}{4}$ with at least 75% of the input bits discarded. This means that the input size needs to be much larger than the output size and there may be a long wait before there is an output (in the event where there is a long stream of 'undesirable' bits of "00"s and "11"s). Thus, despite its excellent de-biasing property, the Von Neumann method is not ideal.

2.3 Compression Based on Good Linear Codes

In this section, we describe a technique proposed by Lacharme [6], which derives good linear compression for random number generation based on good error

correcting codes. The input is a random stream where each input bit has bias e. The output will be a "more" random stream where each output bit has bias $e' < e$.

The method is a generalization of a construction by Dichtl [4]. An example of Dichtl's construction is given by $L : GF(2)^8 \times GF(2)^8 \to GF(2)^8$:

$$L(X, Y) = X \oplus (X \lll 1) \oplus (X \lll 2) \oplus (X \lll 4) \oplus Y.$$

The above function takes 16 independent random bits, each with bias e, and compresses it to 8 bits. In the process, the bias of each compressed bit becomes $2^4 \times e^5$. Thus a RNG which has deteriorated over time, say to give output streams having a bias of 0.05 can be corrected to give a bias of $2^4 \times 0.05^5 = 0.000005$, which is more random. The compression rate is $1/2$ which is the same as XOR: $L(X, Y) = X \oplus Y$. However, XOR only improves the bias from e to $2e^2$.

Thus we see that constructing better linear correctors can achieve better bias while maintaining the same compression rate. Moreover, these compression functions are linear, which makes them efficient to implement on both hardware and software. While Dichtl constructed specific 16 to 8 bit linear correctors, Lacharme generalized his method to apply to more scenarios:

Proposition 1 *([6, Theorem 1]) Let G be a linear corrector mapping n bits to m bits. Then the bias of any non zero linear combination of the output bits is less than or equal to $2^{d-1}e^d$, where e is the bias of each input bit and d is the minimal distance of the linear code constructed by the generator matrix G.*

The above result can be proved by noticing that each output bit of G is an XOR-sum of at least d input bits each with bias e, and then we apply the well-known Piling-Up lemma [5] to get the resulting bias $2^{d-1}e^d$.

Thus we can deduce by Proposition 1 that the linear corrector $L(X, Y)$ gives output streams with bias $2^4 \times e^5$ because L is the generator matrix of a $[16, 8, 5]$ error correcting code.

3 Comparison of Random Bias of Different Post-Processing Functions

In the existing literature, the idea of using post-processing functions based on linear codes is not new. Examples of this can be found, in [6] and [9], where BCH and extended BCH codes were used to construct linear corrector functions. We note, however, that the particular codes picked in these papers do not offer significant advantages over the Von Neumann corrector. We shall illustrate this point numerically.

The codes $[255, 21, 111]$ and $[256, 16, 113]$ codes were used in [6] and [9] respectively. Suppose that the bias of the input bit stream is 0.25, then the bias and throughput, on applying the $[255, 21, 111]$ corrector, is $2^{110} \times (0.25)^{111} = 2^{-112}$ and $\frac{21}{255} = 0.0824$ respectively. In comparison, we get zero bias and a throughput of $\frac{1}{4} - 0.25^2 = 0.1875$ if we had used the Von Neumann corrector. Clearly, the

corrector based on the $[255, 21, 111]$ code has no advantage over the well-known Von Neumann corrector in these aspects.

The same can be said of the corrector based on the $[256, 16, 113]$ code. With the same input bit stream bias of 0.25, we get an output bias and throughput of $2^{112} \times (0.25)^{113} = 2^{-114}$ and $\frac{16}{256} = 0.0625$. Again, the Von Neumann corrector is better in these aspects.

Linear code based correctors still have their merits despite what the above examples suggest. We shall show that if the linear code is chosen wisely, such correctors can be preferred over other forms of post-processing methods.

Although we can achieve zero output bias with the Von Neumann corrector, the rate is at best $\frac{1}{4}$, which may be inadequate if there are stringent demands on the output bit throughput. For this reason, it may be desirable to use a different corrector that has a better rate if we are willing to tolerate a small bias in the output bits. The other two methods we looked at in the previous section, hashing and linear compression, can both achieve better throughput than Von Neumann compression. However, hashing is an intuitive approach in which we cannot quantify the bias reduction of the output stream and we therefore leave it out in our analysis. We shall concentrate on the comparison between Von Neumann and linear compression method in the rest of our paper.

To recap, let us suppose that the bias of the input bits is e, then we have the following results for the various correctors described thus far:

Table 1. Rate and Output Bias

	XOR	Von Neumann	Linear Code $[n, k, d]$
Rate	$\frac{1}{2}$	$\frac{1}{4} - e^2$	$\frac{k}{n}$
Output Bias	$2e^2$	0	$\leq 2^{d-1}e^d$

We can observe from Table 1 that the rate for the XOR corrector is at least twice that of the Von Neumann corrector, while the output bias for the latter is much better than for the former. However, the weaknesses of these correctors are as outstanding as their strengths. The Von Neumann corrector has a low throughput of at most $\frac{1}{4}$, while the XOR corrector does not improve the output bias very significantly. It is desirable to construct a corrector that offers a better trade-off between the rate and the output bias. We shall demonstrate that with proper choices of n, k, and d, the Linear Code (more specifically, BCH code) corrector fulfills this purpose.

In Table 2, we have compared the rate and output biases for the XOR, Von Neumann, linear correctors based on various BCH codes with $n = 255$ and different input bias values, e. The numbers in square brackets are in the usual $[n, k, d]$ notation used to represent BCH codes. The linear codes we have chosen are such that they produce a throughput greater than that of the Von Neumann method. Table 2 lists the rate and output bias for values of $e = 0.25, 0.1, 0.01$ and 0.001.

Table 2. Rate and Output Bias for Various Input Bias, e

$e = 0.25$	XOR	Von Neumann	$[255, 223, 9]$	$[255, 171, 23]$	$[255, 107, 45]$	$[255, 55, 63]$
Rate	0.5	0.1875	0.875	0.671	0.420	0.216
Output Bias	0.0625	0	$\leq 2^{-10}$	$\leq 2^{-24}$	$\leq 2^{-46}$	$\leq 2^{-64}$
$e = 0.1$	XOR	Von Neumann	$[255, 231, 7]$	$[255, 171, 23]$	$[255, 115, 43]$	$[255, 63, 61]$
Rate	0.5	0.24	0.906	0.671	0.451	0.247
Output Bias	0.01	0	$\leq 2^{-17.3}$	$\leq 2^{-54.4}$	$\leq 2^{-100.8}$	$\leq 2^{-142.6}$
$e = 0.01$	XOR	Von Neumann	$[255, 247, 3]$	$[255, 191, 17]$	$[255, 131, 37]$	$[255, 71, 59]$
Rate	0.5	0.2499	0.967	0.749	0.514	0.278
Output Bias	0.0001	0	$\leq 2^{-17.9}$	$\leq 2^{-96.9}$	$\leq 2^{-209.8}$	$\leq 2^{-334.0}$
$e = 0.001$	XOR	Von Neumann	$[255, 247, 3]$	$[255, 191, 17]$	$[255, 131, 37]$	$[255, 71, 59]$
Rate	0.5	0.249999	0.969	0.749	0.514	0.278
Output Bias	0.000001	0	$\leq 2^{-27.9}$	$\leq 2^{-153.4}$	$\leq 2^{-332.7}$	$\leq 2^{-529.0}$

The throughputs for the BCH code correctors vastly outperform that of the XOR and Von Neumann correctors. The output biases are also very much smaller than that of the XOR corrector. In all cases, to just outperform the throughput of Von Neumann corrector, d only needs to be at most 63. If the input bias is small, i.e. 0.1 or 0.01, we can use a code with very low d to obtain a throughput of nearly 1 at the cost of a small output bias of less than 2^{-17}. The near quadrapling of the throughput is a large improvement over the Von Neumann method.

We have shown that with a proper choice of BCH code, a better trade-off between the throughput and output bias can be achieved. Although the output bias is non-zero, it is small enough to be acceptable in some applications.

4 Comparison of Adversary Bias of Different Post-Processing Functions

Sometimes, an adversary might be able to predict the output of a random number generator with probability more than 1/2. The reasons might be due to:

(1) A black box random number generator might be bought from a dishonest vendor, who planted some bugs/backdoor to leak information on the random stream output.
(2) The random number generator designer might make an honest mistake and design a weak RNG, where the output stream is predictable.

Because of the above scenarios, we make the following definition.

Definition 1 *Suppose an adversary can predict the output of a random number generator with probability p_A. Then the adversary bias $e_A = |p_A - 1/2|$.*

4.1 Adversary Bias after Linear Compression

Next we shall show that using linear correctors for random stream compression is better than using Von Neumann compression to lower the adversary bias. First we shall demonstrate the effect with a numerical example.

Example 1. Suppose a random number generator produces the following 16-bit random stream and the adversary knows $12/16 = 75\%$ of the random output:

$$\begin{aligned} \text{Random stream:} &\quad 0101101010001101 \\ \text{Adversary stream:} &\quad 0101\mathbf{01}1010\mathbf{11}1101. \end{aligned}$$

Then Von Neumann on the random stream gives 001110 while Von Neumann on the adversary's stream gives 000110. Now the adversary knows $5/6 = 83\%$ of the compressed stream, the adversary bias actually increase during Von Neumann compression!

If we had used the linear corrector $L(X, Y) = X \oplus (X \lll 1) \oplus (X \lll 2) \oplus (X \lll 4) \oplus Y$, then the random stream compresses to 10101111 while the adversary's stream compresses to 01111011. Now he only knows $4/8 = 50\%$ of the random stream.

□

The reason why the linear corrector outperforms Von Neumann compression when correcting adversary bias is as follows. Let $L(\cdot)$ be the linear corrector, AS denote the output stream known to the adversary and RS be the actual random stream. Also let e_L be the adversary bias after compression and e_A be the adversary bias before compression, then

$$\begin{aligned} e_L &= |Pr(L(AS) = L(RS)) - 1/2| \\ &= |Pr(L(AS) \oplus L(RS) = 0) - 1/2| \\ &= |Pr(L(AS \oplus RS) = 0) - 1/2| \\ &= 2^{d-1} e_A^d \text{ where } e_A = |Pr(AS = RS) - 1/2|. \end{aligned}$$

This computation shows that the reduction in adversary bias is as good as the reduction in random bias for linear compression. However, we showed in the previous example that the adversary bias after Von Neumann compression can become worse (higher), although we have perfect correction for the random bias.

4.2 Adversary Bias after Von Neumann Compression

Von Neumann compression can be deemed as irregular due to the irregularity of the production of outputs. According to the definition of the Von Neumann compression in Section 2.2, we see that there is a probability of $\frac{1}{2}$ where the compression gives no output.

In the worst case assumption, we assume a knowledgeable adversary who is able to have information on the timing when there is no output after Von Neumann compression. This is a possible scenario due to trapdoors or lack of

buffer in the generator, or in side channel attacks where the difference in timing of outputs is analysed. When there is no output from the compression in the middle of the stream, the timing between a first output and a second output is longer due to the missing output in the middle. This constitutes a valid analysis on the Von Neumann output stream.

In these cases, the adversary will be able to guess the nonproduction of output and hence, have auto-correlation in the output streams for comparison. Thus, we will consider this condition in this section. We will also show in Section 4.3 that, in the worst case assumption, linear compression will still outperform Von Neumann compression even if the adversary is resourceful enough to know the timing when no output is produced from the compression.

A theoretical bound on Von Neumann compression can be derived if we consider 3 types of output, namely "0", "1" and "X" (no output). Let $VN(\cdot)$ be the Von Neumann corrector, AS denote the output stream known to the adversary and RS be the actual random stream. Also let e_R be the bias of the random stream and e_A be the adversary bias before compression. Since the bias of the random stream e_R is the bias of the input bits to the Von Neumann corrector, this means that for an input bit x of the random stream,

$$p_R = Pr(x = 0) = \frac{1}{2} + e_R \text{ and } q_R = Pr(x = 1) = \frac{1}{2} - e_R. \tag{2}$$

Similarly for adversary bias e_A and an input bit x' of the adversary stream,

$$p_A = Pr(x' = x) = \frac{1}{2} + e_A \text{ and } q_A = Pr(x' \neq x) = \frac{1}{2} - e_A. \tag{3}$$

Every two input bits will give one output bit through the Von Neumann corrector. Therefore, the probabilities of the adversary predicting the Von Neumann output correctly under the two conditions, $AS = RS$ and $AS \neq RS$, are given in Table 3 below.

Table 3. Adversary Stream Prediction Probabilities

RS input	01	10	00	11	Total probability
AS input ($= RS$ input)	01	10	00	11	p_A^2
AS input ($\neq RS$ input)	-	-	11	00	$q_A^2(p_R^2 + q_R^2)$
VN output	"0"	"1"	"X"	"X"	$p_A^2 + q_A^2(p_R^2 + q_R^2)$

It is clear that if the adversary guessed the input bits correctly, the probability of getting the same output as with the random stream is the same as the probability of guessing the two input bits correctly, which is p_A^2.

$$Pr(VN(AS) = VN(RS) |\ \text{AS} = \text{RS}\ ) = p_A^2. \tag{4}$$

In the case of adversary guessing the wrong input bits, there are only two situations where the adversary gets the same output. That is, when the random

stream input is "00" and the adversary's guess is "11", or vice versa. In this case,

$$\begin{aligned} & Pr\ (VN(AS) = VN(RS)|\ \text{AS} \neq \text{RS}\ ) \\ &= Pr(RS = \text{“11”}, AS = \text{“00”}) + Pr(RS = \text{“00”}, AS = \text{“11”}) \\ &= q_R^2 q_A^2 + p_R^2 q_A^2 \\ &= q_A^2 (p_R^2 + q_R^2). \end{aligned}$$

Hence, the total probability p_V of the adversary guessing the output of the Von Neumann corrector correctly is given by

$$\begin{aligned} p_V &= Pr(VN(AS) = VN(RS)) \\ &= Pr(VN(AS) = VN(RS)|\ \text{AS} = \text{RS}\ ) + Pr(VN(AS) = VN(RS)|\ \text{AS} \neq \text{RS}\ ) \\ &= p_A^2 + q_A^2(p_R^2 + q_R^2) \\ &= (\frac{1}{2} + e_A)^2 + (\frac{1}{2} - e_A)^2[(\frac{1}{2} + e_R)^2 + (\frac{1}{2} - e_R)^2] \\ &= (\frac{1}{4} + e_A + e_A^2) + (\frac{1}{4} - e_A + e_A^2)[(\frac{1}{4} + e_R + e_R^2) + (\frac{1}{4} - e_R + e_R^2)] \\ &= (\frac{1}{4} + e_A + e_A^2) + (\frac{1}{4} - e_A + e_A^2)(\frac{1}{2} + 2e_R^2) \\ &= (\frac{3}{2} + 2e_R^2)e_A^2 + (\frac{1}{2} - 2e_R^2)e_A + \frac{1}{2}e_R^2 + \frac{3}{8}. \end{aligned}$$

As a result, the adversary bias e_V of the output after Von Neumann compression will be

$$e_V = |p_V - \frac{1}{2}| = |(\frac{3}{2} + 2e_R^2)e_A^2 + (\frac{1}{2} - 2e_R^2)e_A + \frac{1}{2}e_R^2 - \frac{1}{8}|. \tag{5}$$

4.3 Linear Compression Outperforming the Von-Neumann Compression

Let us define some notations and also give a summary of the results we have derived at so far.

Table 4. Bias after Compression

	Bias After Von Neumann Compression	Bias After Linear Compression
Random Bias, e_R	$e_{RV} = 0$	$e_{RL} \leq 2^{d-1} e_R^d$
Adversary Bias, e_A	$e_{AV} = \lvert(\frac{3}{2} + 2e_R^2)e_A^2 + (\frac{1}{2} - 2e_R^2)e_A + \frac{1}{2}e_R^2 - \frac{1}{8}\rvert$	$e_{AL} \leq 2^{d-1} e_A^d$

In section 3, it is shown that although the random bias after linear compression is always greater than that after Von Neumann, we can have the random bias lowered to a value close to zero which makes it comparable to the Von Neumann compression. In other words, although $e_{RL} > e_{RV}$ but by choosing a large enough d, we can have $e_{RL} \approx e_{RV}$.

Next we compare the adversary bias after each compression. Figure 1 below illustrates the adversary bias after each compression for the case where $e_R = 0.1$. The modulus graph denoted by e_{AV} shows the adversary bias after Von Neumann compression while the e_{AL} graph shows the adversary bias after linear compression for $d = 3, 4$ and 20.

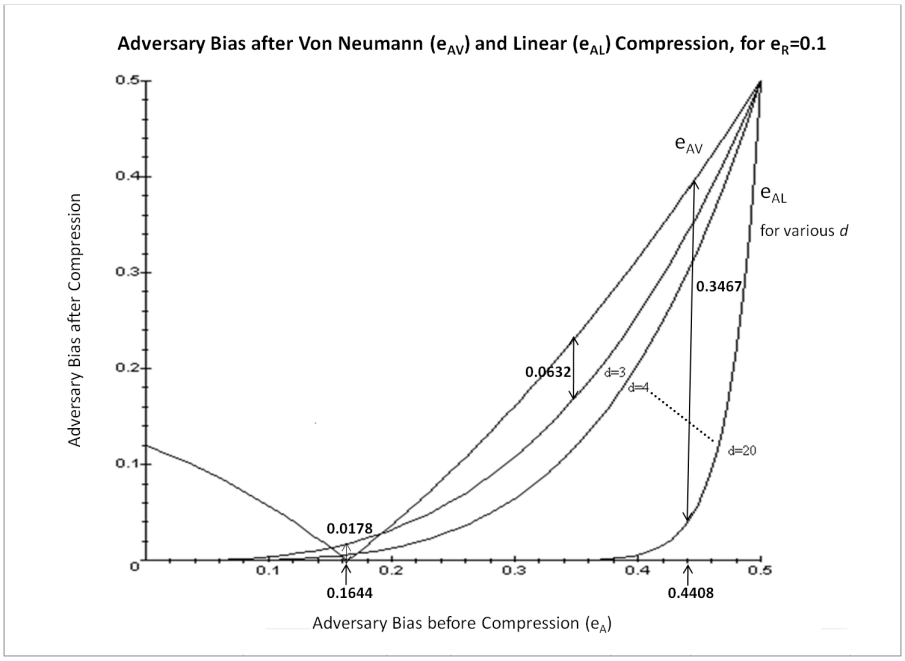

Fig. 1. Impact of compression on adversary bias

Consider the case $d = 3$. The graph shows that linear compression gives a better reduction of adversary bias most of the time except for $0.15025 \leq e_A \leq 0.19219$. Linear compression can outperform Von-Neumann compression (i.e. e_{AL} is smaller than e_{AV}) by as much as 0.0632 while Von-Neumann compression outperforms linear compression (i.e. e_{AV} is smaller than e_{AL}) by at most 0.0178. As d increases, the range of e_A where Von Neumann compression outperforms linear compression will get narrower where its size tends to zero, the advantage of Von Neumann compression in this range will also tend to zero while the advantage of linear compression will increase substantially. For the case where $d = 20$, linear compression is comparable to Von Neumann compression when $e_A = 0.1644$

while it outperforms Von Neumann compression for all other values of e_A. The maximum advantage occurs when $e_A = 0.4408$, where the advantage of linear compression is $e_{AV} - e_{AL} = 0.3869 - 0.0402 = 0.3467$. Table 5 summarizes our discussion and compare the advantage of linear compression for different d.

Table 5. Maximum Advantage of Linear Compression over Von Neumann Compression, $e_R = 0.1$

d	Small Range where Von Neumann outperforms Linear	Max Advantage of Von Neumann	Max Advantage of Linear
3	$0.1502 < e_A < 0.1922$	0.0178	0.0632
4	$0.1594 < e_A < 0.1714$	0.0058	0.1158
5	$0.1625 < e_A < 0.1665$	0.0019	0.1551
6	$0.1638 < e_A < 0.1651$	0.0006	0.1859
⋮	⋮	⋮	⋮
20	$0.1644 - \delta < e_A < 0.1644 + \delta$, δ negligible	10^{-5}	0.3467

Similar results hold for varying values of e_R. Summarising, the linear compression reduces the adversary bias much more than the Von Neumann, except over a negligible range of e_A with d suitably large, say $d = 20$. Moreover, in the unlikely event that the Von Neumann outperforms the linear compression, the large d will reduce the adversary bias to a value close to zero, which makes it comparable to the Von-Neumann . Thus we conclude that the linear compression is much more effective in lowering the adversary bias than the Von Neumann.

4.4 The Use of Linear Codes with Large d

In Section 3, we pointed out that the codes $[255, 21, 111]$ and $[256, 16, 113]$ does not offer any advantage over Von Neumann compression in terms of both throughput and random bias correction. However, it is very effective for correcting adversary bias.

Suppose we have a RNG whose random bias is $e_R = 0.005$, which is not too bad. But it is bought from a dishonest vendor who planted a trapdoor and is able to guess the RNG output with bias $e_A = 0.3$. Using the formulae from Sections 4.1 and 4.2, linear compression with $[255, 21, 111]$ would give adversary bias:

$$e_{AL} = 2^{110} \times (0.3)^{111} = 2^{-82.8},$$

while Von Neumann compression would give adversary bias (Table 3):

$$e_{AV} = (0.8)^2 + (0.2)^2 \times (0.505^2 + 0.495^2) - 0.5 = 0.16.$$

I.e. the probability of the adversary guessing the output is reduced from 80% to very close to 1/2 for linear compression while it is still 66% for Von Neumann compression.

5 Implementation

In this section, we describe two constructions for implementing linear corrector functions based on BCH codes. We also aim to estimate the resources required to implement the linear codes in ASIC technology, as well as provide resource utilization results for our FPGA implementation.

5.1 Construction of Linear Corrector Functions Based on Cyclic Codes

BCH code is a family of cyclic codes with high hamming distance that can be derived from a generator polynomial. As such the generator matrix can be easily derived and the code is efficient in lowering both the random bias and adversary bias when used as a linear compression. A list of generator polynomials and their corresponding parity check polynomials of $[n, k, d]$ BCH codes with $n = 255$ is given in Appendix A for reference.

Either the generator polynomial or the parity-check polynomial can be used in the implementation. Both methods provide efficient hardware implementation. We first describe the generator polynomial method.

For input $X = (x_{n-1}, \cdots, x_0)^T$, the output $Y = (y_{k-1}, \cdots, y_0)^T$ is defined as the product of the generator matrix G and the vector X:

$$Y = \begin{pmatrix} g_{n-k} & \cdots & \cdots & \cdots & g_0 & 0 & \cdots & 0 \\ 0 & g_{n-k} & \cdots & \cdots & \cdots & g_0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \ddots & & \vdots \\ 0 & \cdots & 0 & g_{n-k} & \cdots & \cdots & \cdots & g_0 \end{pmatrix}_{k \times n} \begin{pmatrix} x_{n-1} \\ x_{n-2} \\ \vdots \\ x_0 \end{pmatrix}_{n \times 1}$$

where $g_{n-k}, \cdots, g_0$ are the coefficients of the generator polynomial $g(x)$.

This transformation mapping can be implemented using the circuit shown in Fig. 2 [10]. In the first $n - k + 1$ cycles, the input bits are shifted (MSB first) into a register of $n - k + 1$ bits (shift register 1). In the next k cycles, while the remaining input bits are shifted into shift register 1, the output bits are produced by XOR-ing certain bits of shift register 1 and fed into a register of k bits (shift register 2). The position of the XOR taps is determined by the generator polynomial $g(x)$. In this example of the $[255, 247, 3]$ BCH code with generator polynomial $g(x) = x^8 + x^4 + x^3 + x^2 + 1$, a total of 256 registers and 4 XOR gates are required for the implementation.

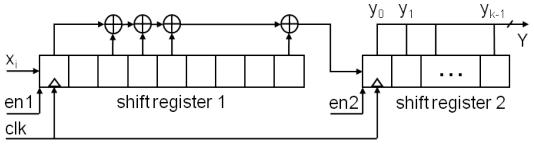

Fig. 2. $[255, 247, 3]$ hardware implementation using the generator polynomial

Next, we describe the parity-check polynomial method, which is a modular polynomial reduction [6]. By definition, parity-check polynomial $h(x) = (x^n - 1)/g(x)$ is at most of degree k. Using polynomial modulo under $h(x)$, the output has at most k bits. The function for this mapping from input vector X to output vector Y is defined by:

$$Y = X \bmod h(x)$$

The circuit for this method is similar to a Galois LFSR, shown in Fig. 3. Again, the position of the XOR taps is determined by the parity-check polynomial $h(x)$. In this example of the $[255, 21, 111]$ BCH code with parity-check polynomial $h(x) = x^{21} + x^{19} + x^{14} + x^{10} + x^7 + x^2 + 1$, a total of 21 registers and 6 XOR gates are utilized.

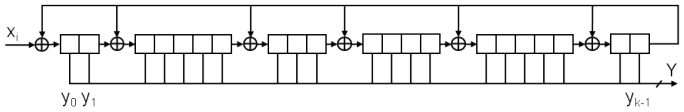

Fig. 3. $[255, 21, 111]$ hardware implemention using the parity-check polynomial

5.2 Resource Utilization

As the linear corrector function can be constructed using either the generator polynomial or the parity-check polynomial, the implementation which uses fewer resources is preferred. In ASIC technology, a rough comparison can be made by estimating the Gate Equivalent (GE) of the resources used. Consider the [255, 247, 3] code in Appendix A. Its generator polynomial is of weight 5 whereas its parity check polynomial is of weight 128. Implementing with the generator polynomial, we need 256 registers and 4 XORs. Suppose a XOR uses 2.67 GE while a register uses 6 GE [11], the total resources required is then 1,546.68 GE. On the other hand, implementing the parity check polynomial will require 1,821.09 GE. The resources for the two different implementations of the codes in Appendix A is calculated and summarized in Table 6 below. As observed in the table, generating the whole codespace using $g(x)$ is better than using $h(x)$, in terms of resources usage, only for the first 3 codes of dimension $[255, 247, 3]$, $[255, 231, 7]$ and $[255, 223, 9]$. Using $h(x)$ for the other codes is better as fewer resources are required for implementation.

In FPGA technology, the same comparison cannot be made easily. This is because different FPGA devices have different logic cell architectures, and FPGA design tools may apply optimizations automatically. Therefore, the more efficient construction to use for the FPGA in interest is best found out through actual implementation.

We implemented the codes in Appendix A on a Xilinx Spartan 3A-DSP XC3SD1800A. The number of slices utilized is shown in Table 7. From the table, we can see that the implementation using the parity-check polynomial uses

Table 6. Comparison of estimated Gate Equivalents for $n = 255$

k	$w_1 = wt(g(x)) - 1$	$w_2 = wt(h(x)) - 1$	Using $g(x)$, # GE $= 6(n+1) + 2.67w_1$	Using $h(x)$, # GE $= 6k + 2.67w_2$	Polynomial requiring less GE
247	4	127	1,546.68	1,821.09	$g(x)$
231	14	111	1,573.38	1,682.37	
223	20	123	1,589.40	1,666.41	
191	38	87	1,637.46	1,378.29	$h(x)$
171	42	85	1,648.14	1,252.95	
131	58	67	1,690.86	964.89	
115	74	57	1,733.58	842.19	
107	72	47	1,728.24	767.49	
71	84	47	1,760.28	551.49	
63	104	31	1,813.68	460.77	
55	92	27	1,781.64	402.09	

fewer resources than that of the generator polynomial for all cases. This can be explained as follows. As the number of XOR gates required is always less than the number of registers, the slice utilization is largely dominated by the number of registers required. Thus, since the parity-check polynomial (k registers) always uses less registers than the generator polynomial ($n + 1$ registers), its slice utilization is also less.

In general, for Xilinx FPGA, the implementation with the parity-check polynomial should utilize fewer resources in most cases. However, this may not hold true for FPGA devices from other vendors. Therefore, as mentioned earlier, the construction which is more resource-efficient for a particular FPGA is best found out through actual implementation.

Table 7. Comparison of FPGA implementation resources for $n = 255$

k	Using $g(x)$, # slices	Using $h(x)$, # slices
247	148	145
231	148	136
223	149	132
191	149	112
171	141	101
131	140	77
115	145	67
107	140	63
71	135	42
63	142	37
55	128	32

6 Conclusion

In this paper, we studied the benefits of using linear compression for post-processing random number generators over the Von Neumann and XOR corrector. We find that when suitable linear codes of dimension $[n, k, d]$ are selected, the random bias and adversary bias can be greatly lowered while maintaining a high throughput. The general idea is to select a d large enough to lower both biases to acceptable values. In our study, we found that for $e_R = 0.1$, $d = 20$ is generally sufficient to reduce both random and adversary bias to approximately zero. Also, the rate given by $\frac{k}{n}$ will be more efficient compared to the Von Neumann corrector. For instance, using $[255, 171, 23]$-BCH code as a linear corrector reduces biases to approximately zero after linear compression with a higher throughput of $\frac{171}{255} \approx 67\%$, compared to Von Neumann compression with zero bias of throughput 24%. A suggested list of BCH codes for $n = 255$, their generator and parity check polynomials are given in the appendix. Implementation issues and results of resource utilization are discussed in the last section.

References

1. NIST Approved Algorithms for Secure Hashing, `http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html`
2. S. Halevi: Cryptographic Hash Functions and their many applications. In: USENIX Security Symposium. (2009) `http://people.csail.mit.edu/shaih/pubs/Cryptographic-Hash-Functions.ppt`
3. J. Von Neumann: Various Techniques used in Connection with Random Digits. In: National Bureau of Standards Applied Mathematics Series 12, pp. 36-38. (1951)
4. M. Dichtl: Bad and Good Ways of Post-Processing Biased Physical Random Numbers. In: FSE 2007. LNCS, vol. 4593, pp. 127-152. Springer-Verlag. (2007)
5. M. Matsui: Linear Cryptanalysis Method for DES Cipher. In: Eurocrypt 1993. LNCS, vol. 765, pp. 386-397. Springer-Verlag. (1994)
6. P. Lacharme: Post-Processing Functions for a Biased Physical Random Number Generator. In: FSE 2008. LNCS, vol. 5086, pp. 334-342. Springer-Verlag. (2008)
7. A. J. Menezes, P. C. van Oorschot and S. A. Vanstone: Handbook of Applied Cryptography. CRC Press. (1997)
8. M. Grassl: Code tables: bounds on the parameters of various types of codes. `http://www.codetables.de`
9. B. Sunar and D. Stinson: A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. IEEE Transactions on Computers, vol. 56, No. 1. (2007)
10. D. Schellekens, B. Preneel and I. Verbauwhede: FPGA Vendor Agnostic True Random Number Generator. In: International Conference on Field Programmable Logic and Applications. (2006)
11. A. Poschmann: Lightweight Cryptography - Cryptographic Engineering for a Pervasive World. (2009) `http://eprint.iacr.org/2009/516.pdf`

A Appendix: BCH Codes

Table 8. Generator and Parity Check Polynomials of BCH Codes

Code Parameters $[n, k, d]$	Generator Polynomial $g(x)$	Parity-check Polynomial $h(x)$
$[255, 247, 3]$	11D	8E25C0C93720ADACB0FB7 AE886C79CC5A452A7767B F4CD460EABE509FE178D
$[255, 231, 7]$	1BBA1B5	E7400884547D0D3D1A82 98CB0B2497ECD4CAD9 60F1F70B471667357EE5
$[255, 223, 9]$	1EE5B42FD	CA1E95439F31F12A925E 61E1BF5DE175C668DA 9B159BC7A1F93FF66D
$[255, 191, 17]$	16CE707E26B6F9977	BF5F0B83A04CF7C58047CBF0 B8C8A5D8E28C2C4609020D6B
$[255, 171, 23]$	1B0E46229C4EE1F8C7319F	E3E79B7AFE3243AA9A400A CAB2138885EBF0B40BBE3
$[255, 131, 37]$	11BCB6CCE6906958 AA17F2231050EB39	8D493EDCA6106BE2A FA4F85A2A3DE6879
$[255, 115, 43]$	1855B6B7A2029D679E 826017CEAB732E75DF	FDD74802A09D1F 88718D4B97B1EA3
$[255, 107, 45]$	1242FE9A4365732A1EC 04EB9E207EBE7A0D921	905F8D71982A80 0DE9456B55D21
$[255, 71, 59]$	140A722A1A468D36D87A2536 4E685922A1E56FD1A478C1D	AAFC7EA6FFFBEF0A8D
$[255, 63, 61]$	11EC9E8B4E7646AB351EEFE38 0F6C49EB4B56F8BD770AC6C1	8FC22EFA9CC296C1
$[255, 55, 63]$	1D9B1541D04805B06AF58C1A1 635618D6F6822DE248B076778F	D466E1C119DAB3

The generator polynomials $g(x)$ and parity-check polynomials $h(x) = (x^n - 1)/g(x)$ are represented in hexadecimals such that in the binary form, the rightmost bit represents the coefficient of the constant term 1 and the leftmost non-zero bit represents the degree of the polynomial.

Example 2. The generator polynomial $g(x)$ for BCH code $[255, 247, 3]$ is represented as 11D in Table 8. In binary form, hexadecimal 11D is $(000100011101)_2$, which represents $x^8 + x^4 + x^3 + x^2 + 1$ in polynomial form. The rightmost bit represents the coefficient of the constant term (x^0) and the leftmost non-zero bit is in the 9th position from the right, representing the degree of the polynomial as 8.

□