3D Visualization Applied to PRBGs and Cryptography – Long Version

Michel Dubois, Eric Filiol∗

Operational Cryptology and Virology Laboratory, 38 Rue des Docteurs Calmette et Guérin, 53000 Laval, France

Abstract Today there is no easy and quick way to analyse and differentiate random data. However, all day long our computers generate pseudo random data, our cryptographic algorithms tend to act as pseudo random generator of data to better hide the message. So we can then ask whether is it possible to quickly determine the algorithm used to construct a random sequence of numbers and in a second time, distinguish between a PRBG or a cryptographic algorithm? In this paper, we present a new approach, to visualize, in a two and three dimensions environment at the same time, a sequence issued from a pseudo random bit generator or from cryptographic algorithms. To embody our idea, we assume that sequences produced by PRBG and Cryptographic algorithms are comparable to a nonlinear system generating a chronological series of data. We have developed some tools to realize our analysis and use them to well known kinds of PRBG and to the AES. Even, if our approach can’t serve as determining proof on the quality of an alea, it can bring a great help to quickly (because visually) distinguish two random sequences and eventually find some statistical bias.

Keywords Block Cipher, Boolean Function, Cryptanalysis, AES

1 Introduction

In computer security, visualization is commonly used for different tasks like log analysis [16], attacks detection [13], binary analysis [4] and reverse engineering [3, 2] but today there is no easy way to analyze and differentiate random data. However, operating systems or cryptographic protocols commonly use randomness, for example, to generate a TCP sequence number or generate a random encryption key for WiFi or Web.

Likewise, the Grail of any cryptographic algorithm is to obtain, at each internal step and after the encryption process, a sequence appearing to be the closest possible of perfect random. Indeed, the security of a cryptographic algorithm depends on its ability to generate unpredictable quantities. Assuming that the perfect randomness is only a philosophical view and, in fact, the perfection of the randomness depends on statistical tests were applied to him [5], we can say the cryptographic randomness must be random in the sense that the probability of a particular value chosen must be low enough to prevent an opponent to gain the advantage by optimizing a search strategy based on this probability [12].

So we have pseudorandom number generator algorithms and cryptographic algorithms which producing data that seem random. The challenge is multiple. First of all, how to quickly determine the algorithm used to construct a random sequence of numbers. Then, is it possible to distinguish between a PRBG or a cryptographic algorithm. Finally, if we have a cryptographic algorithm, is it possible to use visualization to realize a first approach of its security.

This paper is composed as follows. In Section 2, pseudorandom bit generators are described. In Section 3, methodologies to display linear sequence in 2 and 3 dimensions space are presented. In Section 4, we use 2D and 3D visualizations to analyze some pseudorandom bit generator sequences. In Section 5, we apply our approach on the RC4 and AES algorithms. Finally, in Section 6, conclusions and future directions are provided.

2 Pseudorandom Bit Generator

A random bit generator is a device or an algorithm that produces a statistically independent bit sequence and unbiased [12].

Some hardwares generate randomness from the time between the emission of particles during radioactive decay phase or from the thermal noise from resistance or semiconductor diode. Similarly, to generate randomness, some software use algorithms combining various sources such as the time between two keystrokes, mouse movement or content of input/output buffers.

A pseudorandom bit generator (PRBG) is a deterministic algorithm which, from a truly random sequence of bits of length \( k \), produces a sequence of bits of length \( l \gg k \) seemingly random. The initial sequence is called the seed while the sequence produced by the PRBG is called pseudo-random bit sequence.

Generate random number is a difficult problem for computers because they are deterministic and as said by John Von Neumann “Any one who considers arithmetical methods of producing random digits is, of course, in a state of
The sequence produced by a PRBG is not truly random. More specifically, the number of possible sequences produced by the PRBG is, at most, a small fraction, namely $\frac{2^k}{k}$, of all possible binary sequences of length $l$. The objective here is to take a small truly random sequence and extend to a sequence of much greater length, so that the attacker cannot easily distinguish between the output sequences of PRBG and truly random sequences of even length.

For a PRBG algorithm to be cryptographically secure, three main rules have to be respected [12].

Firstly, the length of the seed $k$ must be of sufficient size. In fact, the size of $k$ must be such that research on all $2^k$ seeds space elements is impossible for the attacker.

The following rule is that the sequence produced by a PRBG must be statistically close to a truly random sequence, or more accurately, approximated by a sequence of binary variables, independent and identically distributed. We then say that PRBG spends all statistical tests in polynomial time if no polynomial algorithm can correctly distinguish between a sequence produced by the PRBG and a truly random sequence of the same length with probability $p \gg \frac{1}{2}$.

Finally, the product elements should not be predictable, from a partial sequence already known, to an attacker with limited computing resources. A PRBG respect this rule, so-called “following bit” if, from the first $l$ bits of a sequence $s$ produced by the PRBG, no polynomial algorithm is able to predict the $(l+1)$ bit of $s$ with probability of $p \gg \frac{1}{2}$.

### 3 PRBG representations

To better understand the results obtained from different PRBG algorithms, we will represent the sequences obtained in a 2D and 3D environment simultaneously.

#### 3.1 2D representation

A PRBG is comparable to a nonlinear system generating a chronological series of data. If we want to represent such a series in a two-dimensional environment, a first approach could be to browse linearly all the points of the plan by assigning to each dot a color corresponding to an entry in the series. This idea seems good, but it has the drawback of not represent the reality of the series.

Indeed, if we take a plan delimited by a rectangle of width $x$ and high $y$ with $x \times y = |n|$, $|n|$ representing the cardinal of elements of the series $n$, so the dot at the coordinates $(i,j)$, representing the $n(t)$, $t < |n|$ member of the series, has as neighbors the points $(i - 1,j)$ and $(i + 1,j)$ representing respectively the $n(t - 1)$ and $n(t + 1)$ members of the series, but also the points $(i,j - 1)$ and $(i,j + 1)$ representing the $n(t')$ and $n(t'')$ members of the series. So points on the same line correspond to elements that follow one in the series but not between lines. As an example we can see on the figure 1 that the 15th node has as neighbors the 14th and 16th nodes but also the 8th and 20th node that are not close from it in the sequence.

Thus, to obtain a better representation of our series in a two-dimensional space we need another algorithm. A space filling curve is a continuous one-to-one function which map a compact interval to a multi-dimensional unit hypercube[6]. Space filling curves were discovered by the mathematician Giuseppe Peano in 1890[15]. For our purposes, we take out a specific example of such a curve, the one proposed by David Hilbert[7] shortly after Peano’s discovery.

A Hilbert Curve is a space filling curve which maps a one-dimensional interval into a two-dimensional area. Its construction is based on the repetition of a simple pattern: the first three sides of a square. At each repetition the square is turned, reduced and repeated until to obtain a curve that fills the plan. In fact, a Hilbert curve can be regarded as a Lindenmayer system[19], also known as an L-system. A L-system is a string rewriting system that can be used to generate fractals with dimension between 1 and 2. For the Hilbert curve the L-system rules are:

$$
L = +RF - LFL - FR+
$$

$$
R = -LF + RFR + F L-
$$

with $L$ and $R$ as L-system alphabet, $F$ means draw forward, $-$ means turn left 90 and $+$ means turn right 90.

The first orders of the Hilbert curve are presented in the figure 2. As we can see on the pictures 2a, 2b and 2c, each point has an index corresponding to the index of each entry in the series. We can see that unlike our first approach described above, the use of Hilbert Curve permit us to preserve data locality, meaning that points close in the series remain close in the two-dimensional space.

So, through this presentation mode, in two dimensions, we have a first approach to identify a PRBG algorithm.
3.2 3D representation

After the two-dimensional approach, we will try to represent the sequences obtained in a 3D environment.

As we have said above, a PRBG is comparable to a nonlinear system generating a chronological series of data. One of the most commonly used means for analysing a series of this type is to rebuild its phase space using the method of delays [9]. The phase space is a space of \( n \) dimensions that completely describes the state of a \( n \) variables system. For example, the phase space describing the landing of a rocket is a two-dimensional space. The first dimension is the velocity of the rocket and the second dimension is the ground distance. The phase space is then a graph representing the velocity on x-axis and the ground distances on y-axis. Thus, for the rocket to land smoothly, it is necessary that the curve describing the progress in its phase space tends to zero (see figure 3).

\[
x[t] = s[t-2] - s[t-3] \\
y[t] = s[t-1] - s[t-2] \\
z[t] = s[t] - s[t-1]
\]

Then, representing the point sequence thus obtained in a three-dimensional environment we obtain a specific shape to the given function of PRBG. This form, called attractor, reveals the complex nature of the dependencies between the different elements of the sequence generated by the algorithm investigated [22, 23].

As a concrete example, the sequence of numbers of the figure 4 comes from the PRBG algorithm that generates the TCP session sequence numbers of the operating system GNU/Linux RedHat in its version 7.3.

\[
3281499104, 3271545868, 3287443610, 3238749981, 3274168813, 3302234066, \\
3229771300, 3287970591, 3295595222, 3298841199, 3292774952, 3294591612, \\
3294540537, 3294046036, 3296969037, 3293746299, 3300112100, 3292483752
\]

Figure 4. Sample sequences of Redhat 7.3 PRBG

At first glance, it is difficult to determine whether there is a link between the different elements of this sequence of numbers. By cons, the figure 5 formalizes this relationship by unveiling a shape that is characteristic of PRBG used by the Linux 2.4.18 kernel used by this Linux distribution.

Through this presentation mode, we are now easier to identify a PRBG algorithm, given that the same algorithm will always give the same attractor.

4 PRBG samples

We will now study some PRBG algorithms.

4.1 True alea

Before starting our comparison of PRBG, we need a reference, that is to say the representation of the attractor of a true alea. As a computer is a deterministic engine, it is not possible to it to produce a true alea. Only hardware equipment based on physical random elements can provide this type of alea.

The website www.random.org provides the ability to generate sequences of true alea. It uses atmospheric noise to produce this alea. This site is the result of a scientific project of Dr. Mads Haahr from the “School of Computer Science and Statistics” from Dublin Trinity College. It is used for online games, to generate the random lottery games, for science projects... We generated a sequence of 10 000 random integers from this site. The first inputs of this sequence are as presented in figure 6.

And the 3-dimensional representation of the corresponding attractor is shown in figure 7.
and $z$ in blue. The distribution of points in this space forms a homogeneous cloud and evenly covers the volume along the three axes. No specific pattern emerges. The values provided are between 0 and 99995 with an average value of 54707.9. In the right corner we have the Hilbert curve representation of the data set. The colors in this curve seem completely random.

In addition, the program we use, assigns a specific color to each point: all the colors of the color palette is spread over all the points in chronological order. Thus, the first item on the list receives the first color in the palette, the second point receives the second color and so on until the last point. This principle allows us to add a fourth dimension to our graph: the time.

On our curve the color distribution seems completely random.

### 4.2 The Pi digits

The number Pi is a mathematical constant defined by the ratio of a circle’s circumference to its diameter. Pi is commonly approximated as 3.14159265. Being an irrational number, Pi cannot be expressed exactly as a fraction. The digits of Pi appear to be randomly distributed, however no proof of this has been discovered[21].

If we take the 100 first digits of Pi we obtain the sequence presented in figure 8. And the 3-dimensional representation of the corresponding attractor is shown in figure 9.

$$x_{n+1} = ax_n + b \pmod{m} \quad | \quad n \geq 0$$

The integers $a$, $b$ and $m$ are the parameters that characterize the generator and $x_0$ is the seed. An example of generated sequence with $a = 5$, $b = 3$ and $m = 4096$ is given below:

$x_{n+1} = 5x_n + 3 \pmod{4096}$ function is shown in figures 12 and 13.
772, 3863, 2994, 2385, 3736, 2299, 3306, 149, 748, 3743, 2334, 3481, 1024, 1027, 1042, 1117, 1492, 3367, 454, 2273, 3176, 3595, 1594, 3877, 3004, 2735, 1390, 2857, 2000, 1811, 866, 237, 1188, 1847, 1046, 1137

Figure 11. Sample sequences of linear congruence generator

The sample size is 16384, the minimum is 0 and the maximum 4095 the average value is 2047.50.

The color distribution in the Hilbert curve seems random.

But, unlike true random generator that we have taken as a reference, the distribution of points in figures 12 and 13 has a specific pattern in the form of lines aligned on several levels. These lines correspond to the period of the algorithm.

Finally, we note that only the red predominantly colors appear, it means that only the end of the spectrum is visible. The first points are covered by the latest, the algorithm therefore provides several times the same values.

This type of generator is predictable and is not suitable for cryptographic use. Indeed, from a partial sequence, and without knowledge of the parameters $a$, $b$ and $m$, it is possible to reconstruct the rest of the sequence. Moreover, in this case, the both representation – in two and three-dimensional – permit us to refine the randomness property of this algorithm.

### 4.4 Nonlinear recurrence generator

For our next example, we use a nonlinear recurrence generator. The latter produces a pseudorandom sequence of integers $x_1, x_2, x_3, ...$ according to the following recurrence relation [20]:

$$x_{n+1} = \lambda x_n(1 - x_n) \quad | \quad n \geq 0$$

This suite called “logistic” leads, if $\lambda > 3.56095$, to a chaotic suite. Logistic suite is used to model the size of a biological population over generations [11]. An example of generated sequence is given below:

45109885224, 451068728783, 451038599037, 451008465983, 450978329622, 450948189954, 450918046977, 450887900692, 450857751097, 450827598193, 450797441977, 450767282451,

The attractor for this function with the following variables $\lambda = 3.8$ and $x_0 = 0.7364738523$ is given in figure 14.

The figure 14 also shows a specific pattern that makes it an unsafe PRBG for cryptographic application. However, unlike the linear congruence generator, we have no more appearance of periodic pattern. The generated data are not random in the first iterations (green color) then appear to become more random so with the increase of the number of iterations.

In this case, studying the Hilbert curve is interesting because the color distribution seems random at the beginning but becomes quickly uniform at the end of the list (same color).

In our example, the sample size is 16384, the minimum is 18426892666.96 and the maximum value 948850856936.47 sample average is 631015962902.33.

### 4.5 Blum-Blum-Shub generator

The pseudorandom bit generator Blum-Blum-Shub is a computationally secure PRBG as factoring large prime numbers remains insoluble. This generator produces a pseudorandom bit sequence in accordance with the following algorithm:

- generate two great number of Blum-Blum $p$ and $q$ and compute $n = pq$. A Blum-Blum’s number is a prime number congruent to 3 modulo 4;
choose a seed $s$ in the range $[1,n-1]$ such that \( \gcd(s,n) = 1 \);

- compute $x_0 = s^2 \pmod{n}$;

- the sequence is defined as $x_{i+1} = x_i^2 \pmod{n}$;

- if $x_i$ is even then $z_n = 0$ and if $x_i$ is odd then $z_n = 1$;

- the output of the sequence is $z_1, z_2, z_3, \ldots$

The attractor of this function is shown in figure 15. To facilitate the implementation of our presentation, we did not apply the final step of recovering the least significant bit. Finally, we note that the attractor obtained is spread over the three axes in the form of a homogeneous cloud. We find the same form as in the case of the true alea. Actually, the Blum-Blum-Generator is cryptographically secure assuming the intractability of the quadratic residuosity problem [1].

In our example, the sample size is 16384, the minimum is 56782881.0 and the maximum value is 512462853845.0, the sample average is 257668663192.64.

5 Cryptographic algorithms

5.1 RC4 attractor

To calculate the attractor of the RC4 algorithm, we start from the sequence of integers $n = 2^{105} + i$ with $i = [1 \ldots 60 \ 000]$ that we convert in 128-bit blocks. We then encrypt each block with the same encryption 64-bit key:

$$k = 01010101010101$$

Finally, we convert the result to integers. So we start from a linear sequence of integers to get a pseudorandom sequence of integers. The following list shows the result for the first 8 entries in the sequence:

- $RC_{4k}(2^{105} + 1) = 16 \ (001715085553813117608731931814448065)$
- $RC_{4k}(2^{105} + 2) = 16 \ (001715085553813117608731931814448064)$
- $RC_{4k}(2^{105} + 3) = 16 \ (001715085553813117608731931814448063)$
- $RC_{4k}(2^{105} + 4) = 16 \ (001715085553813117608731931814448062)$
- $RC_{4k}(2^{105} + 5) = 16 \ (001715085553813117608731931814448061)$
- $RC_{4k}(2^{105} + 6) = 16 \ (001715085553813117608731931814448060)$
- $RC_{4k}(2^{105} + 7) = 16 \ (001715085553813117608731931814448059)$
- $RC_{4k}(2^{105} + 8) = 16 \ (001715085553813117608731931814448058)$

We developed a suite of specific tools (see Appendix A.2) to generate the list and then display it in a three dimensional environment like the approach outlined above for PRBG. We get the figure 16.

The attractor of RC4 is very sparse and boils down to three red balls. The color is important, it is the color of the end of the panel color. So we can deduce that all points of the attractor have the same coordinates. We are far from a random distribution of the space like with the Blum-Blum-Shub generator.

If we change the key we obtain different kinds of attractor: figure 17 and figure 18. So we have a close link between the obtained attractor and the key used. The Hilbert curve representation of RC4 shows the same link between the attractor and the key used but more important it shows that this algorithm does not seems really close to the perfect randomness.

5.2 Enigma machine attractor

The Enigma machine was invented by German engineer Arthur Scherbius in 1927 [8]. The Model A was quickly abandoned in favor of the model B, of the size of a typewriter then by a portable version equipped with lamps indicators with the model C. The Enigma machine and the company Scherbius founded for marketing, will vegetate during the inter-war. The Wehrmacht buys some copies for evaluation. Its only when Hitler began to rearm Germany than the cryptology experts of the Wehrmacht decide to adopt it and to equip the German army.

The Enigma machine is a polyalphabetic encryption machine. Its first version has 3 rotors each with 26 positions [12]. The $R_1$ rotor rotates every time a key is pressed,
the $R_2$ rotor rotates according to the movement of the $R_3$ that acts like an odometer. The movement of the rotor generates a different combination each time a key is pressed. The encryption key is defined by the initial position of the three rotors.

To calculate the attractor of the Enigma machine, we use a sequence comprising all possible combinations of 5 characters from AAAAA to EZZZZ. That is to say $5 \times 26^4 = 2284880$ inputs. Each of them is then encrypted using a program reproducing operation of a three rotors Enigma machine. The used key is AAA. The obtained result is then converted to hexadecimal. Thus the sequence AAAAA becomes FTZMG ⇒ (0x46545a4d47) and EZZZZ becomes DXILI ⇒ (0x4458494c49).

The attractor of the Enigma machine in a 3-Dimensions world is presented in the figure 19 page 177.

The analysis of the attractor of the Enigma shows a cloud of random points. However, we distinguish alignments and a cloud of scattered enough points. Since there are more than two million points we should have a denser cloud.

By zooming in the graph (figure 20 page 177), we are seeing that points are grouped in clusters and there colors are similar. By counting the points inside clusters we find they contains approximately 26 points, ie a point for each character of alphabet. We have no explanation for this fact, it needs more investigations.

### 5.3 AES attractor

To calculate the attractor of the AES, we start from the sequence of integers $n = 2^{105} + i$ with $i = [1 \cdots 1\,000\,000]$ that we convert in 128-bit blocks. We then encrypt each block with the same encryption key:

$$k = 80000000000000000000000000000000$$

Finally, we convert the result to integers. So we start from a linear sequence of integers to get a pseudorandom sequence of integers. The following list shows the result for the first 8 entries in the sequence:

<table>
<thead>
<tr>
<th>Entry</th>
<th>AES($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d5576f0b48727b4b8a46765</td>
</tr>
<tr>
<td>2</td>
<td>f235b44131b61671286376116112889045</td>
</tr>
<tr>
<td>3</td>
<td>568b01d27a2002007b700f876</td>
</tr>
<tr>
<td>4</td>
<td>6f223e8776955312133674928760278999499</td>
</tr>
<tr>
<td>5</td>
<td>48490b9b9a8718e0912ab46</td>
</tr>
<tr>
<td>6</td>
<td>72555455556238525213433946355100516077</td>
</tr>
<tr>
<td>7</td>
<td>6c864c6c4a27a9a9978494</td>
</tr>
<tr>
<td>8</td>
<td>5d936a6475420008126664105441210386598198</td>
</tr>
</tbody>
</table>

By using the same tools as for the RC4 algorithms, we get the figures 21 and 22.
Instead of the Blum-Blum-Shub generator we find that the attractor of AES takes the form of a cloud of points distributed over the three axes under the form of cubes. It differs, in this, from the attractor of the true alea. Nevertheless we can reasonably infer that the AES is an encryption algorithm designed as a pseudorandom bit generator close to the true hazard.

After graphically analysed the behaviour of the AES, we will try to see how the algorithm behaves with different encryption keys. For this, we have reproduced the process described above using the same initial sequence and encrypting it with the two keys:

\[ k_1 = \begin{array}{c}
\begin{array}{c}
80000000000000000000000000000000
\end{array}
\end{array} \]  \hspace{1cm} (3)

\[ k_2 = \begin{array}{c}
\begin{array}{c}
40000000000000000000000000000000
\end{array}
\end{array} \]  \hspace{1cm} (4)

These two keys differ only on 1 bit over the 128 bits composing them.

Then assigning a different color to each attractor, we can compare them two by two. The result is shown in figures 23 and 24. To generate these figures we calculated the encryption of a sequence of 45000 identical blocks.

By performing the comparison of these two attractors, we find that they fit together tightly by uniformly covering the three dimensions. However, it is not possible to distinguish a possible correlation between these two sets of elements. To help visualize a possible correlation between the two sets of encrypted blocks, we will represent the same graph but by joining two by two the corresponding points of the same input. Thus, the point representing the encrypted block with the key \( k_1 \) of the first block of the sequence will be linked to the corresponding point in the encrypted block with the key \( k_2 \) of the same block in the clear.

The result is shown in figures 25 and 26. In the aims that the graphs are readable, we have reduced the size of the sample to 255 entries. The straight path connecting the points is messy and shows no identifiable pattern.

We can reasonably deduce that there is no correlation between the encryption of the same clear block with different keys and they are sufficiently independent.
Correlation between two AES attractors - axe $z \rightarrow z'$

calculation of Minkowski distances \[18\] with $p = 2$ for each point of the graph confirms this result. Indeed, the list of distances, whose first inputs are shown below, behave as a random sequence. The attractor arising from the representation of these distances in three dimensions (see figure 27) has the form of a cloud whose specific form is close to that a diamond. It would be interesting to try to understand what justifies this form, but it is not the subject of this work.

12026847709294981120, 12026869209288865792, 19971070890283577344, 26113153353646714880, 25972774069680922624, 14954846745435789312, 7391670208430327808, 1673631091276342528, 2444035869612603904, 2802878496097698304, 2426843600751804928, 1753123651095763712, 9810861268161230848, 10326031878050832384, 16430236778313869312, 14587535298828013568, 14873569053264308224, 13174990125624895488

Attractor of distances between two encryption process with the AES - view $45^\circ$

6 Conclusion

Our analysis of the hazard produced by PRBG and in particular by the cryptographic algorithms, although visually speaking, is not sufficient to prove that the sequences produced by its algorithms are statistically random.

However, viewing of the hazard, as we have shown, has two attractive advantages: the first is that it can detect patterns and thus here some problems since we are seeking randomness, and second is the fact that our brain has more ability to quickly distinguish two different pictures, our approach allows us to more easily recognize and differentiate PRBG from another. Thus, just by displaying in environments in two and three dimensions the output of a random sequences, we have the opportunity to detect possible patterns and identify the PRBG used to generate the sequence. This approach also allows us to realize a first and quick analysis of potential biases on unknown cryptographic algorithms.

We developed some tools that offer the opportunities to apply our approach on some PRBG and cryptographic algorithms. Our aim is to implement more and more of these algorithms to construct a database of attractors that can be used to attribute the proper PRBG or cryptographic algorithm to a random number sequence.

Acknowledgements

The authors wish to thank Alexey Zhukov of the Bauman Moscow State Technical University for his valuable comments to this research work.

A Listings of programs

The entire program code developed during this article is available on GitHub (https://github.com/archoad). These programs are all under GPL v3.

A.1 PRBG generating and viewing in 3D

Available at the link https://github.com/archoad/PRBG-3D. To run this suite of programs requires a C compiler and OpenSSL and OpenGL libraries.

A.2 Encryption algorithms and 3D visualization

Available at the link https://github.com/archoad/VisCipher3d. To run this suite of programs requires a C compiler and the OpenSSL libraries (using the libcrypto for encryption and the BIGNUM API) and OpenGL.

REFERENCES


