Introduction

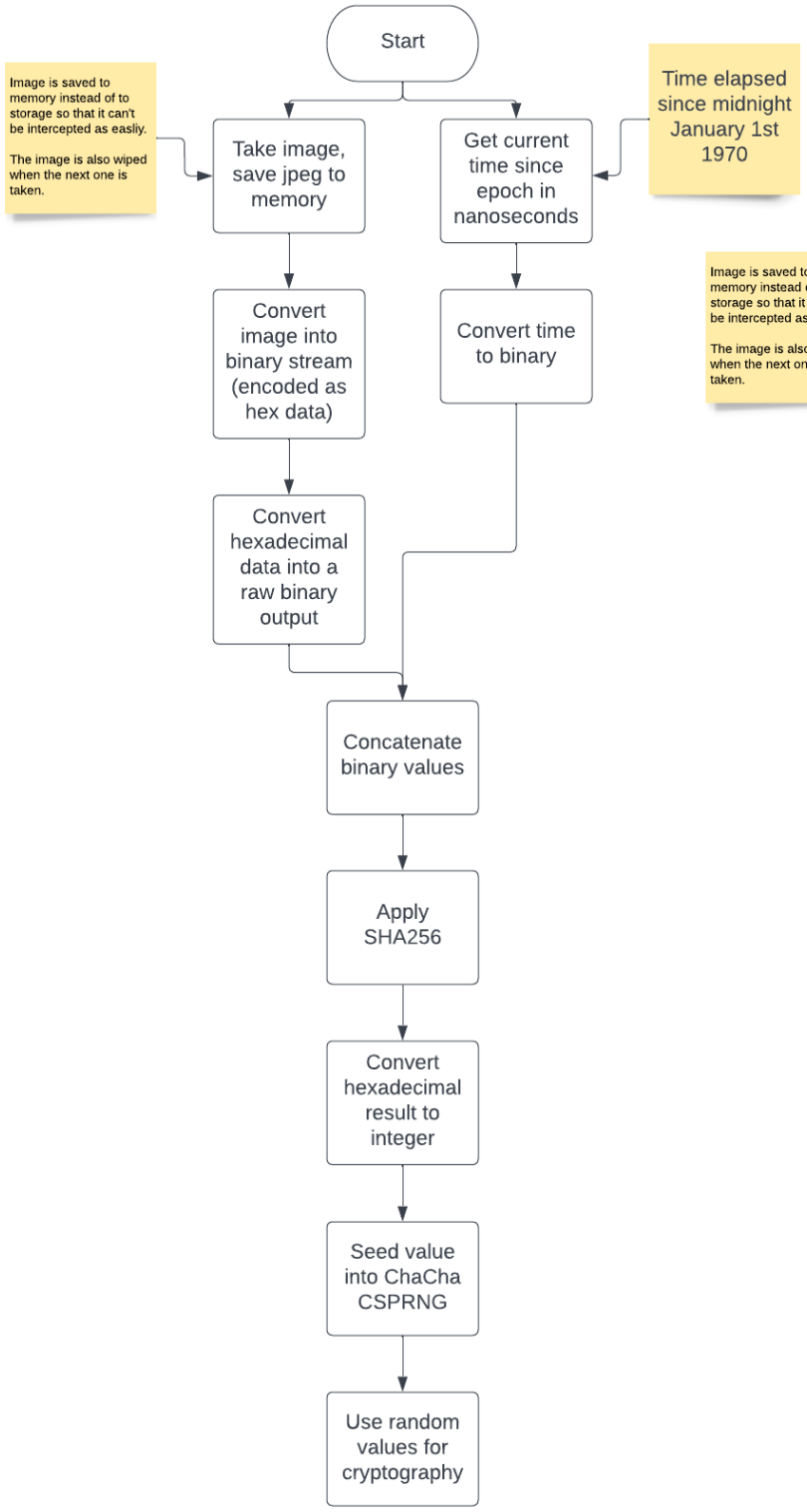
At the heart of cryptography is random numbers. Almost all encryption methods require randomly generated keys to function securely. “In cryptography, the term random means unpredictable.” (Liebow-Feeser, 2017a). If an attacker is able to predict the keys used for encrypting a message, the message may as well not be encrypted at all.

There are 2 types of randomness in computing, true randomness and pseudorandomness. True randomness is obtained by measuring a physical process, it must be unpredictable even to the person measuring it. The most common example is radioactive decay because “the decay of an unstable nucleus is entirely random in time so it is impossible to predict when a particular atom will decay”(Wikipedia Contributors, 2022b).

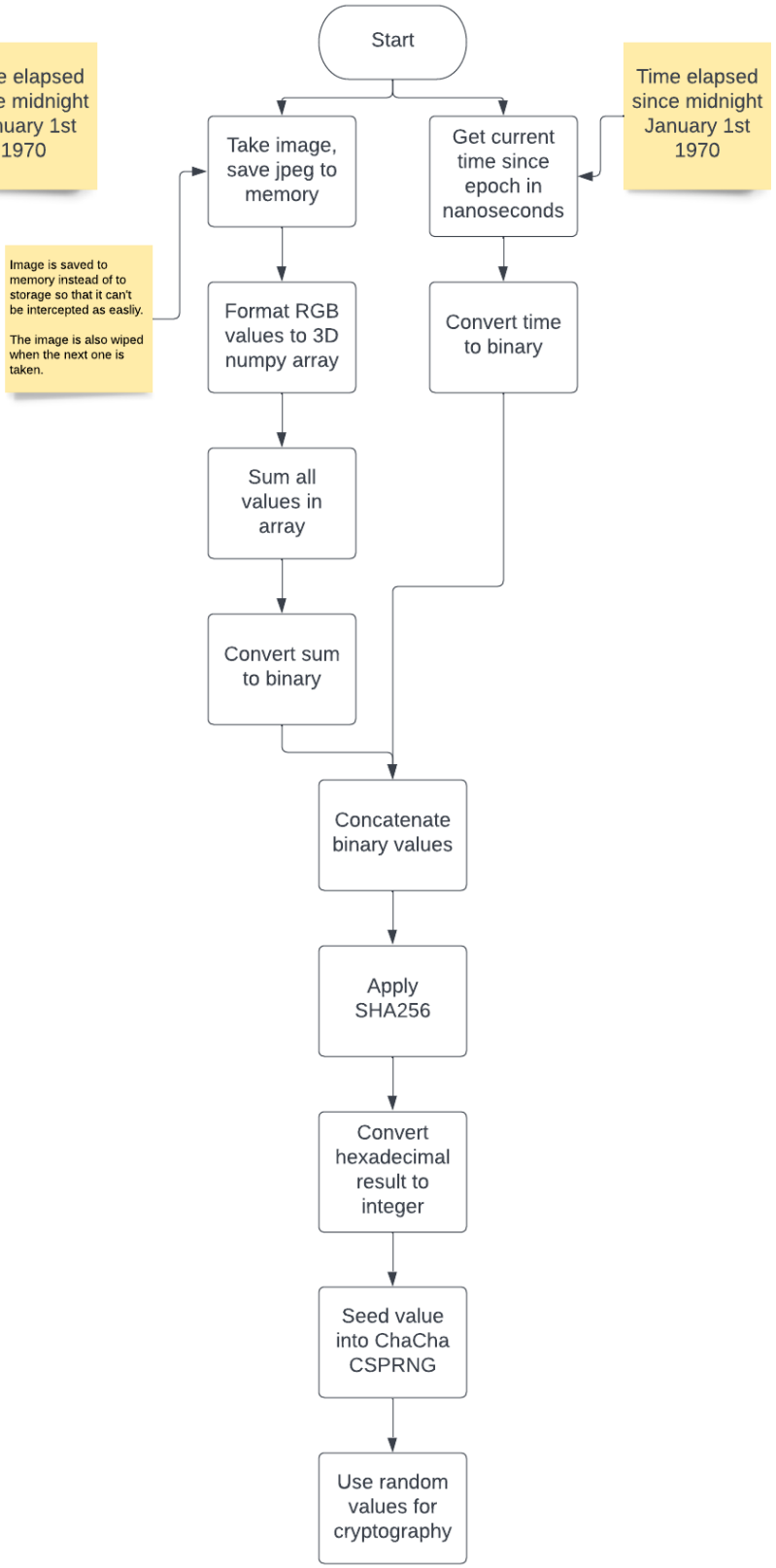
Using true randomness for cryptographic applications is often not practical because of how expensive and time-consuming it is to obtain. Instead, pseudorandomness generated by a deterministic algorithm is used. An algorithm is considered deterministic if given the same input it always produces the same output, a common example is a hash function. When given a random input known as a seed, these algorithms are called cryptographically secure pseudorandom number generators or CSPRNGs. If an attacker is unable to predict the seed, they will also be unable to predict the output of the algorithm.

System flowchart

Standard method



Alternative method



High level overview

My approach is based on the Lavarand design (patent US5732138A) created by Silicon Graphics in 1996. After the patent expired in 2016 the design was used by Cloudflare in their San Francisco office for generating random numbers for encryption on the web.

The design involves pointing a camera at a chaotic system and extracting the data from it. Mixing the data with another entropy source (not covered in original patent). Applying a hash function to the data and seeding the result into a cryptographically secure pseudorandom number generator.

Extracting image data

I have come up with 2 methods for extracting information from the captured images in order to produce the seed for the random number generator.

The first method captures the raw binary data from the image. The JPEG images use 24-bit colour depth producing 145711104 (2464 x 2464 x 24) bits of entropy.

The second method sums the RGB values of each pixel, this is an interesting approach as a 2464 x 2464 image would require 18213888 individual numbers to be added producing a total that is hard to predict.

Although both are valid approaches, I think the first method would be the most secure. Addition is commutative and associative meaning numbers can be added in any order (or grouped in any way) and produce the same result. This is problematic because pixels with the same RGB values would sum to the same total regardless of their position in the image. Using the raw binary data on the other hand would create a unique output every time if the input from the camera was unique. This makes the first approach more secure.

Despite there being a clear winner, I have still researched both methods of extracting data from the images and included their implementations in the rest of the report.

Mixing entropy sources

It is possible that if an attacker had physical access to the system they could modify the frames output by the camera. This would cause a single point of failure in the system. To counter this, it is recommended that one mixes multiple entropy sources together.
The second source of entropy I chose is the current time. This is measured in nanoseconds since Unix Epoch (1st January 1970). Mixing this with my existing entropy source means that not only would the attacker have to manipulate the image, they would have to do so to nanosecond precision. This idea is explored later when investigating possible flaws in my system.

Hashing the entropy pool

After concatenating the 2 entropy sources together, the system applies a cryptographic hash function, specifically SHA-256. Since this algorithm is collision-resistant (always producing a different output given a different input) an attacker would have to control both entropy sources in order to predict the output of the hash function. As long as one source remains uncompromised, the system stays secure.

Using pre-made CSPRNG

The Plasma globe itself is not being used to generate the random numbers, instead it is creating a seed to be used in an existing CSPRNG. The commonly used Golden Rule of cryptography is “don’t roll your own!”. This is because modern encryption methods are so complex and contain many technical details that implementing them yourself can result in security flaws. The best practice is to use reputable and audited crypto libraries that have been tried and tested.

Initially, I planned to use the default random number generator built into Python’s Random module. However, the algorithm it uses (Mersenne Twister) is not cryptographically secure. This is due to the fact that it uses linear recursion and therefore “observing a sufficient number of iterations allows one to predict all future iterations.” (Wikipedia Contributors, 2022a)

The random number generator I chose to use in place of this is called ChaCha. ChaCha is considered cryptographically secure when using a sufficient (by default, 20) number of rounds. It is also a deterministic CSPRNG meaning it will produce the same output every time if seeded with the same input. But due to the mixing of multiple entropy sources and the fact that the hashing algorithm is collision resistant, the same seed should never be generated twice.

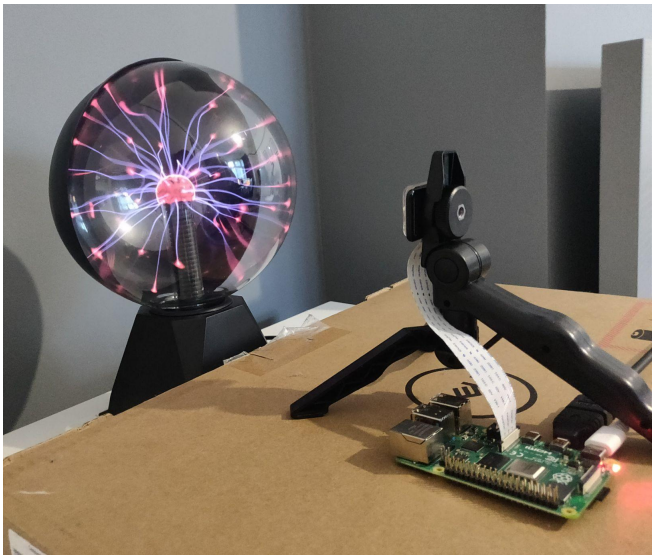
In a real world-production environment, where the security of this system would be critical, I would seed the entropy into `/dev/random`. This is the default CSPRNG built into Linux that implements the ChaCha algorithm. From my research it is considered to be best practice in the server space and is likely far more efficient than a Python program.

Technical explanation

The movement of the plasma inside a plasma ball is a very chaotic and unpredictable system. This works perfectly for producing entropy.

Hardware

I have set up a Raspberry Pi v2 camera module on a tripod pointing at the plasma globe. The camera is connected to a Raspberry Pi 4B via a ribbon cable.

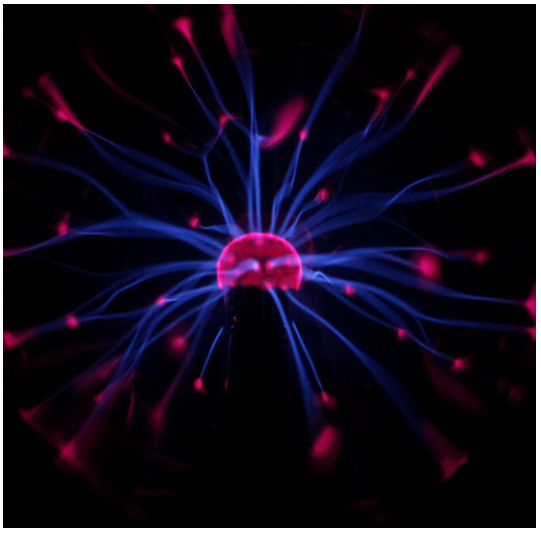


When in use, the plasma ball is set up in a dark room in order to minimise reflections on the glass sphere, this ensures that the entropy is coming only from the plasma itself as well as sensor noise.

Capturing the image

The camera uses an 8 megapixel Sony IMX219 sensor which is capable of capturing images up to 3280 x 2464 in resolution. Images of higher resolution contain more pixels which can be used for generating the entropy. My program creates images that are 2464 x 2464 pixels, giving the largest possible size while retaining a square shape to fit the plasma globe perfectly in frame.

Sample image:



Processing the image data

1 - Using raw binary data

```
with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (2464, 2464)
        camera.capture(output, 'rgb')
        pil_im = Image.fromarray(output, mode="RGB")
        hexStream = BytesIO()
        pil_im.save(hexStream, format="jpeg")
        hex_data = hexStream.getvalue()
        print(hex_data)
```

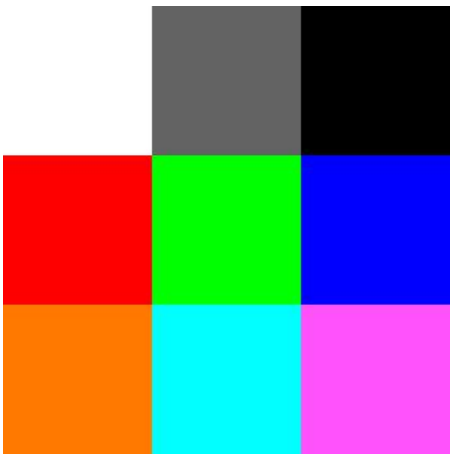
This script captures the image and saves the output as a bytes object, it then produces a large hexadecimal string from the image.

2 - Using pixel RGB values

```
with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (2464, 2464)
        camera.capture(output, 'rgb')
        pixelTotal = numpy.sum(output.array)
        print(pixelTotal)
```

Pixels in an image each contain a value between 0-255 for Red, Green and Blue (RGB). This snippet captures the image from the camera and formats the RGB values for each pixel into a 3 dimensional numpy array. It then uses the `numpy.sum()` method to add up all of the values to produce a total.

Example:



```
[[[255 255 255]
  [100 100 100]
  [  0   0   0]]

 [[255   0   0]
  [  0 255   0]
  [  0   0 255]]

 [[255 120   0]
  [  0 255 255]
  [255  82 255]]]
```

In this example, each row is made up of 3 pixels.

The total sum for all of these values is 3307.

Mixing the image data with a second entropy source

1 - Using hexadecimal image data

```
mixedEntropy =
str(bin(int.from_bytes(hex_data,byteorder=sys.byteorder))[2:]) +
str(bin(time.time_ns())[2:])
```

This line converts the hexadecimal data into a long binary stream, it is then concatenated with the binary value of the current time since epoch measured in nanoseconds.

2 - Using RGB total

```
pixelTotal = int(str(pixelTotal)[1:])
mixedEntropy = str(bin(pixelTotal)[2:]) + str(bin(time.time_ns())[2:])
```

This script begins by removing the first digit of the RGB total, this increases the variability in the values (explained here). Next the binary value of the pixel total is found. This is then concatenated with the binary value of the current time since epoch measured in nanoseconds.

Creating the seed from the entropy and seeding it into the CSPRNG

```
hashHex = hashlib.sha256(str(mixedEntropy).encode('ASCII')).hexdigest()
hashInt = int(hashHex, 16)
randomNum = numpy.random.Generator(ChaCha(seed = hashInt))
randomValue = randomNum.standard_normal()
```

The binary value for the mixed entropy pool is then passed through a SHA-256 hash function producing a long hexadecimal value which is then converted into an integer. This integer is then seeded into the ChaCha CSPRNG using 20 rounds to generate the final random number.

Final scripts

Using binary image data

```
import picamera
import picamera.array
import numpy
from PIL import Image
import hashlib
import time
import sys
from randomgen import ChaCha
from io import BytesIO

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (2464, 2464)
        camera.capture(output, 'rgb')
        pil_im = Image.fromarray(output.array, mode="RGB")
        hexStream = BytesIO()
        pil_im.save(hexStream, format="jpeg")
        hex_data = hexStream.getvalue()
        mixedEntropy =
str(bin(int.from_bytes(hex_data,byteorder=sys.byteorder))[2:]) +
str(bin(time.time_ns())[2:])
        hashHex = hashlib.sha256(str(mixedEntropy).encode('ASCII')).hexdigest()
        hashInt = int(hashHex, 16)
        randomNum = numpy.random.Generator(ChaCha(seed=hashInt))
        randomValue=randomNum.standard_normal()
        print(randomValue)
```

Using pixel RGB totals

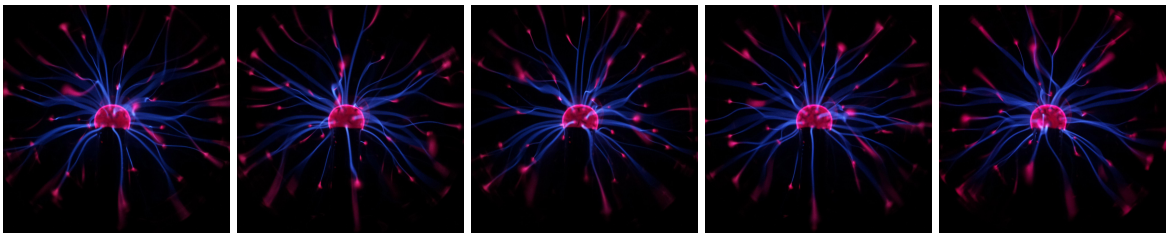
```
import picamera
import picamera.array
import numpy
from PIL import Image
import hashlib
import time
from randomgen import ChaCha

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (2464, 2464)
        camera.capture(output, 'rgb')
        pixelTotal = numpy.sum(output.array)
        pixelTotal = int(str(pixelTotal)[1:])
        mixedEntropy = str(bin(pixelTotal)[2:]) + str(bin(time.time_ns())[2:])
        hashHex = hashlib.sha256(mixedEntropy.encode('ASCII')).hexdigest()
        hashInt = int(hashHex, 16)
        randomNum = numpy.random.Generator(ChaCha(seed = hashInt))
        randomValue = randomNum.standard_normal()
        print(randomValue)
```

Testing the predictability of the system

In order for the CSPRNG to work the best, the value seeded into it needs to be truly unpredictable. To test this I took 1 image per second for 100 seconds and calculated the RGB totals for each image. I then imported these values into a spreadsheet to analyse the results.

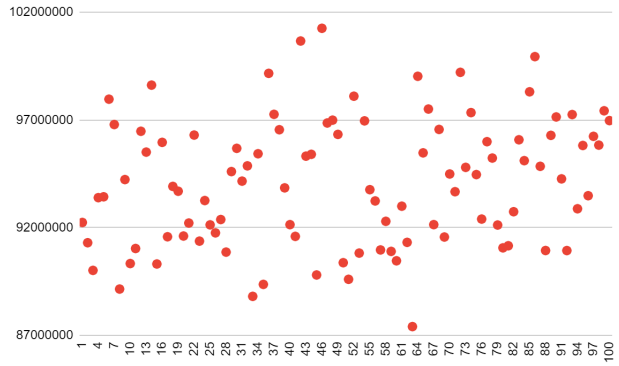
Sample of the first 5 images and their RGB totals

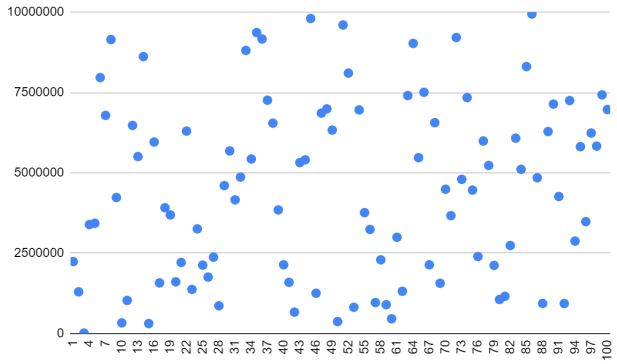


92239410	91298307	90015152	93391126	93431448

As you can see all of the values are in the 90 million range - the first digit is the same for each. This characteristic was reflected in 92/100 images. One could argue because of this that the outcome is quite predictable.

To combat this, I removed the first digit of each value leaving only less significant digits. This increased the variance in the values significantly as shown in the following 2 graphs.

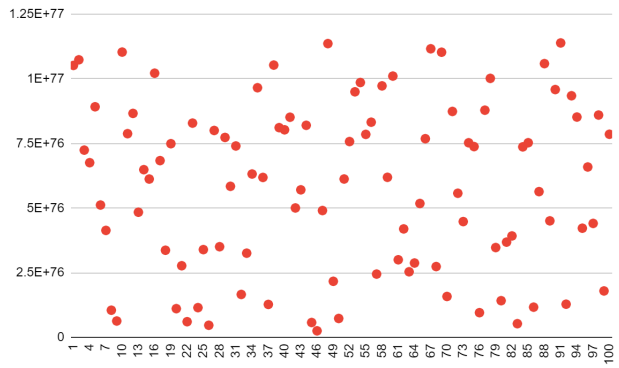


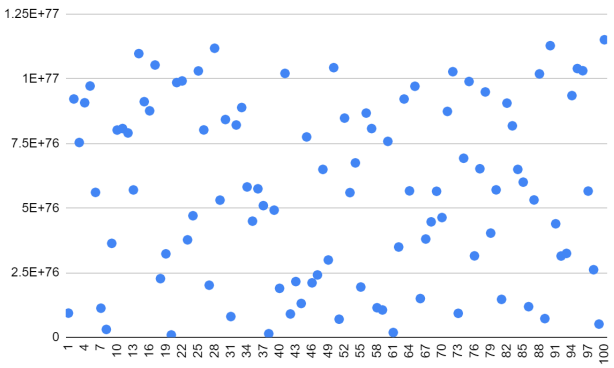


The first (red) graph represents the values as calculated by the script. The second (blue) graph contains the values after removing the first digit. It is clear that the second graph has a greater spread of data. Removing the first digit from the RGB totals makes the values less predictable.

The effect of hashing the data

Removing the most significant digit to reduce predictability was an interesting observation to make; however the final program is hashing the entropy values using SHA-256 before seeding them into the CSPRNG. This defeats the need to remove the first digit from the total. Here are the graphs of the same values after hashing.





Both datasets being equally spread out proves the robustness of the SHA-256 algorithm by showing that even with a somewhat predictable input, the algorithm will sufficiently spread the output data.

It visualises 2 of the 4 main characteristics of a good hash function.

1. “The hash function "uniformly" distributes the data across the entire set of possible hash values.”
2. “The hash function generates very different hash values for similar strings.” (Red graph)
   (SparkNotes, 2022)

Potential attacks against the system

Influencing the Plasma globe

As far as I am aware, it is not possible for an attacker to predict the chaotic state of the Plasma globe if it is in a closed system. However, if an attacker had physical access to the globe they could manipulate the electromagnetic field around it or more simply turn it off. Even if this had occurred, unpredictable entropy caused by the sensor noise would still influence the final random number.

Manipulating the camera output

If an attacker were able to control the complete output from the camera either by predicting the RGB values or spoofing the image entirely, it would still be hard to compromise the system. The camera entropy is mixed with the epoch time in nanoseconds meaning the attacker would have to carry out an exploit with nanosecond precision.
If this system were to be deployed in the real-world, I would mix in even more entropy sources such as keyboard timings, mouse coordinates or CPU temperature. Each of these sources would need to be controlled in order to compromise the whole system.

Reference list

Google.com. (1996). *US5732138A - Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system - Google Patents*. [online] Available at: https://patents.google.com/patent/US5732138 [Accessed 18 Jul. 2022].

Knight, N. (2010). *Safe mixing of entropy sources*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/3429519/safe-mixing-of-entropy-sources?rq=1 [Accessed 20 Jul. 2022].

Liebow-Feeser, J. (2017a). *LavaRand in Production: The Nitty-Gritty Technical Details*. [online] The Cloudflare Blog. Available at: https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/ [Accessed 18 Jul. 2022].

Liebow-Feeser, J. (2017b). *Randomness 101: LavaRand in Production*. [online] The Cloudflare Blog. Available at: https://blog.cloudflare.com/randomness-101-lavarand-in-production/ [Accessed 18 Jul. 2022].

SparkNotes. (2022). *Hash Tables: Hash Functions | SparkNotes*. [online] Available at: https://www.sparknotes.com/cs/searching/hashtables/section2/ [Accessed 14 Aug. 2022].

Vedantu (2020). *Properties of Addition*. [online] VEDANTU. Available at: https://www.vedantu.com/maths/properties-of-addition [Accessed 14 Aug. 2022].

Wikipedia Contributors (2022a). *Mersenne Twister*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Mersenne_Twister [Accessed 6 Aug. 2022].

Wikipedia Contributors (2022b). *Radioactive decay*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Radioactive_decay [Accessed 15 Aug. 2022].