Truly Random Encryption Key Generator

Personal Project

*Sean-David Brokke*

*S&ES  
09-06-2024*

# Introduction

In today's digital world, it's really important to use strong encryption to protect sensitive information. A secure encryption system relies on a random encryption key, because if it's predictable, attackers can crack it. But traditional random number generation methods can have limitations, which could make them vulnerable to attackers.

This personal project is all about tackling this challenge by developing a truly random encryption key generator prototype. By delving deep into the fascinating world of randomness and exploring various advanced random number generation techniques, this project is about creating a secure and reliable solution for real-world cybersecurity applications.

# Research

This chapter covers all the research that has been done on various methods & functions that were used or considered for implementation of the project.

## Pseudo Random Number Generator

Pseudo Random Number Generator(PRNG) refers to an algorithm that uses mathematical formulas to produce sequences of random numbers. PRNGs generate a sequence of numbers approximating the properties of random numbers. A PRNG starts from an arbitrary starting state using a seed state. Many numbers are generated in a short time and can also be reproduced later, if the starting point in the sequence is known. Hence, the numbers are deterministic and efficient.

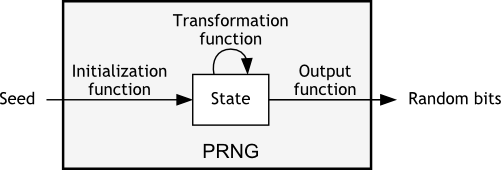


Figure 1: PRNG's function visualized.

### Security

Standard PRNGs (e.g., rand(), std::mt19937) are not designed for cryptographic use and can be vulnerable to attacks if used in security-sensitive contexts. These algorithms are efficient but ultimately deterministic, meaning the sequence can be recreated if you know the starting seed (initial value).

## True Random Number Generator

True Random Number Generator (TRNGs) are devices or processes that generates random numbers based on unpredictable physical phenomena, as opposed to algorithms or mathematical formulas. These unpredictable physical sources provide a higher level of randomness compared to traditional methods, making them crucial for various security applications.

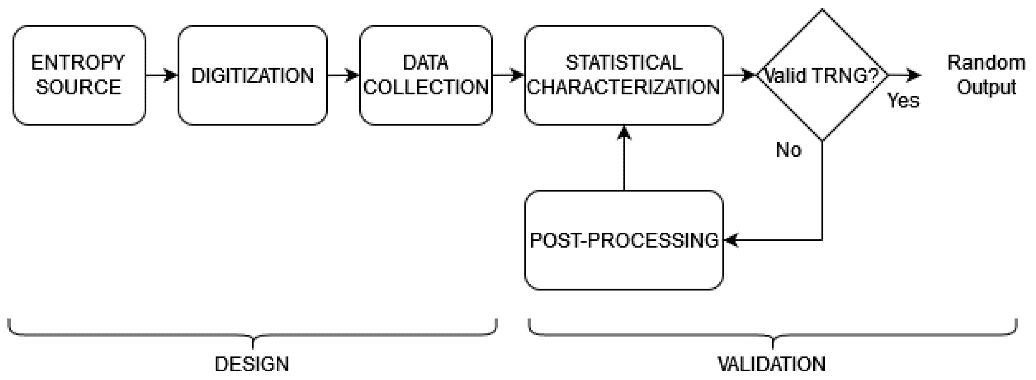


Figure 2:TRNG function example visualized.

### Security

TRNGs rely on physical processes like thermal noise, atmospheric noise, or radioactive decay. These natural phenomena are inherently unpredictable, making it difficult (or impossible) to guess the next random number.

## freeRTOS

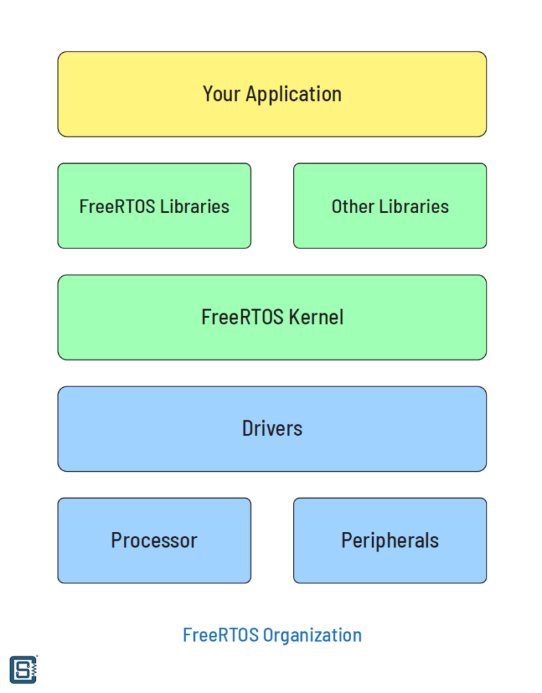
FreeRTOS is a lightweight Real-Time Operating System (RTOS) designed for low-performance processors like microcontrollers. It is an open-source project widely adopted for commercial and industrial applications. An RTOS is a type of operating system with deterministic behavior. An RTOS has a kernel, which handles timing, queuing, task synchronization, priorities, interrupts and more. FreeRTOS can work on a single-core or multi-core environment. In a multi-core environment, tasks can be executed in different cores. If there is only one core, FreeRTOS divides the processor time into small units and allows each task to use them at regular intervals. This means we can run multiple RTOS tasks on the same core without worrying about conflicts.

Figure 3: freeRTOS Organization.

### Synchronization

In the world of concurrent programming, where multiple tasks or processes share resources and communicate with each other, it's important to make sure everything runs smoothly. This is where the idea of task synchronization comes in. It stops things getting out of control, where tasks might access and modify a shared resource at the same time, which could lead to unexpected results. To achieve task synchronization, two main techniques are used: mutexes and semaphores.

#### Mutex

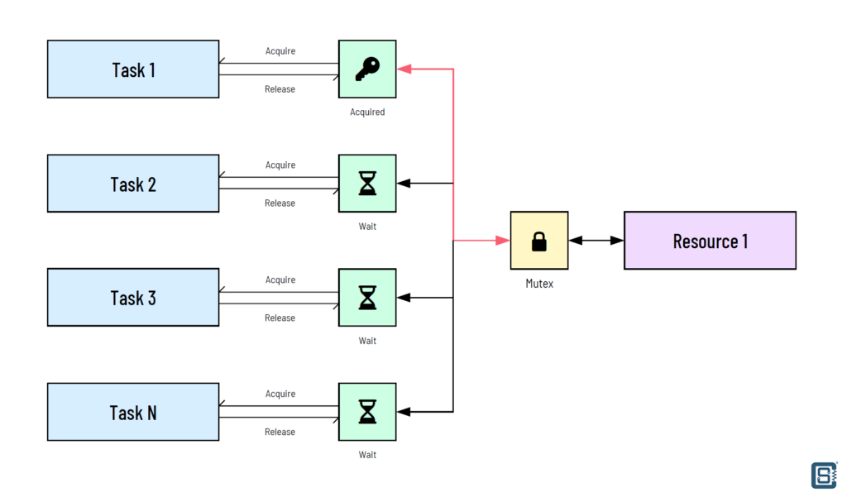
A mutex works like a lock on a resource. Only one task can get the lock (mutex) and access the resource at a time, while other tasks waiting to get to the lock are forced to wait until the lock is released. This ensures that only one person at a time can access the resource and that data is kept safe, especially when making changes to critical data.

Figure 4:Mutex function visualized.

#### Semaphore

A semaphore is different from a mutex because it lets a set number of tasks access a shared resource at the same time. Just picture a limited number of tokens representing access rights. Tasks get these tokens (semaphores) to access the resource. If there are enough tokens available, multiple tasks can go ahead at the same time. This is useful for managing resource pools, like a fixed number of worker threads accessing a larger pool of tasks.

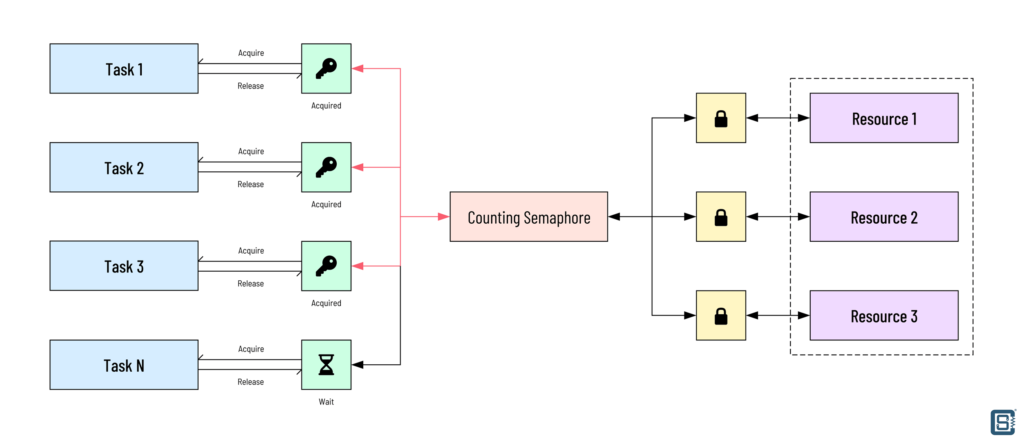


Figure 5:Semaphore function visualization.

## Hashing

A hash function is a mathematical function that takes inputs and transforms them into fixed-length strings of characters. The input to the hash function can be any length, but the output is always the same length. This process creates a unique "fingerprint" for each input. If the input is slightly different, the fingerprint will be different too. This is called "collision resistance".

Hash functions are used in many security applications, including password storage, digital signatures, and data integrity checks. Hash values are the results of a hash function.

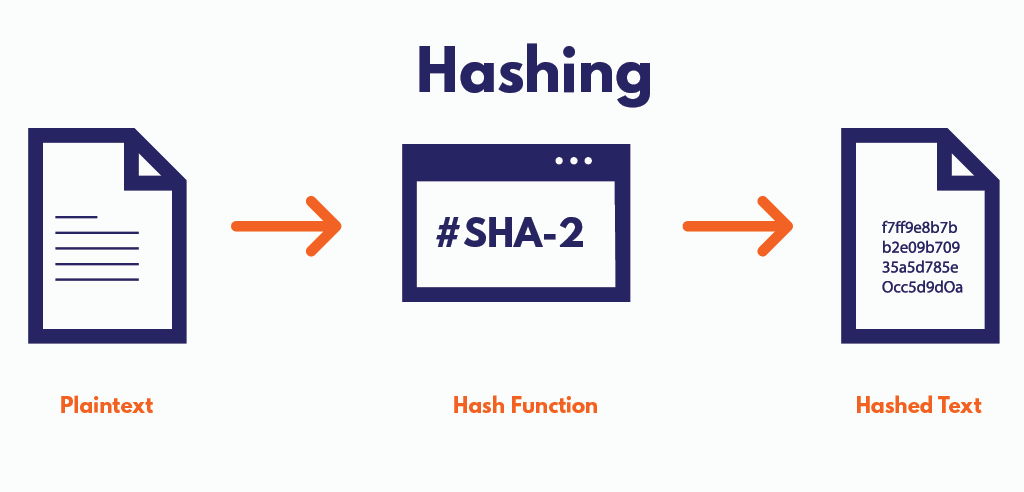


Figure 6: Hash function visualized.

# Implementation

This chapter looks at the technical side of the project, focusing on the functions, methods and key design choices implemented.

We'll look at how the research & ideas we talked about earlier are put into practice in the code to achieve the project's goal. We'll go over the specific functions and methods used, as well as the thinking behind the design decisions made during development.

## TRNG Method

### Pursuit

Generating truly random numbers in software, especially in C++ without relying on hardware support, is challenging. While it's possible to create algorithms that aim to mimic randomness (PRNG), true randomness is difficult to achieve due to the deterministic nature of computers.

The sequence of numbers generated by ‘rand()’ function depends totally on the seed value it’s initialized with. If you use the same seed value, you'll get the same sequence of numbers every time. Even if you update the seed using ‘srand()’ from a different thread, the sequence of pseudo-random numbers generated by ‘rand()’ will still be deterministic. Changing the seed can alter the sequence of numbers produced, but it does not make them truly random.

My initial plan was to go with the option mentioned above, using 1 thread to update the seed while the other generates the random number making them independent from each other, so knowing I can’t achieve truly random number with that method defeated the purpose.

### Discovery

But after doing research on ESP32 which was my MCU (Micro Controller Unit) of choice, I discovered that the ESP 32 board has a built-in hardware random number generator (RNG) module that generates high-quality random numbers based on physical processes.

**Entropy Source:** The hardware RNG uses thermal noise or electronic noise, to generate random bits. These physical processes are naturally unpredictable, which means the numbers generated are completely random.

## Design

### Choice

After all my research and discoveries, I decided on a final prototype design, this prototype will use an esp32 to generate truly random number using its hardware RNG, the random number generated will then be used to generate a random image on an 8x8 led matrix block

Then a camera will use the led block as it’s entropy source, and based on the image on the led block the that the camera captures an encryption key will be generated in python, slight imperfections during image capturing will simply add an extra layer of external physical randomness to the project.

### Truly Random Led-block Image Generator

#### Wiring Diagram

A circuit board with wires connected to it

Description automatically generated

Figure 7: Wiring diagram.

Table 1: Pin connections.

|  |  |
| --- | --- |
| ESP32 | LED-block |
| VIN | VCC |
| GND | GND |
| GPIO23(MOSI) | DIN |
| GPIO5(SS) | CS |
| GPIO18(SCK/VSPI\_CLK) | CLK |

#### UML Diagram

A screenshot of a computer

Description automatically generated

Figure 8: UML Diagram Led block Display.

#### State Machines

A diagram of a computer

Description automatically generated

Figure 9: State Machine for main\_app loop.

A diagram of a random image

Description automatically generated

Figure 10: State Machine for the generate\_random\_image\_task.

A diagram of a computer

Description automatically generated

Figure 11: State Machine for the display\_image\_task.

#### Code

Since the Arduino framework doesn't natively support FreeRTOS for ESP32 development, utilizing the ESP-IDF framework became necessary.

A screen shot of a computer program

Description automatically generatedIDisplay Interface  
The IDisplay interface provides the necessary functions for initializing and displaying images on a display.

Since ESP-IDF’s primary language is C, ‘\_\_cplusplus’ guards are necessary for implementing cpp code. C++ allows for function overloading, which lets you define multiple functions with the same name but C does not support this so in order to prevent name mangling these guards are needed.

Figure 12: IDisplay Interface.

##### A computer screen with text on it Description automatically generatedLedBlockDisplay Class

This Class inherits it’s functions from the IDisplay interface, the private attributes consists of SPI pins for setting hardware pins, SPI host for configuring SPI settings and max7219 which is a library for the integrated circuit (IC) that drives the Led block.

Figure 13: LedBlockDisplay header file.

A screen shot of a computer program

Description automatically generatedIn the ‘init()’ function has a bunch of log messages those were added to help aid during troubleshooting /debugging process.

To set up the SPI bus, we create a spi\_bus\_config\_t structure called buscfg. This holds all the settings for the SPI interface. We then set the attributes for the pins we're using: ‘.mosi\_io\_num’ to ‘PIN\_NUM\_DIN’, and‘.sclk\_io\_num’ to ‘PIN\_NUM\_CLK’.

As we're not using the other SPI functionalities the corresponding attributes needs to be set to -1 to indicate they're not used. Unused flags are set to 0, and max transfer size is set to 32(max size).

Figure 14: SPI config.

After initializing the ‘max7219\_t’ object attribute ‘dev’ in the ‘init()’ function.

In the LedBlockDisplay class, the display() function takes an image pointer and displays it on the LED block. It gets the actual image data from the memory location that's pointed to by image. Then the max7219\_draw\_image\_8x8 function is called to translate the image data and control the LED block using the initialized dev object.

A screen shot of a computer program

Description automatically generated

Figure 15: Display function.

##### A computer screen shot of code Description automatically generatedgenerate\_random\_image\_task

Figure 16:generat\_random\_image\_task.

function generates random 8x8 images. It does this by creating a 64-bit integer ‘new\_image’ and filling it with random 8-bit values ‘(esp\_random() & 0xFF)’ for each row of the 8x8 matrix. Then, it updates the shared variable `pCurrentImage` with the current random image. We use a mutex `xImageMutex` to make sure that only one task can access this shared variable at a time. This stops other tasks from interfering with each other. The task runs forever, creating a new random image every second, logging it, and then releasing the mutex before waiting to create the next image. This makes sure that any updates to the shared image variable are thread-safe. And the reason for choosing mutex over semaphore is because we are only sharing one resource not multiple.

##### A screen shot of a computer program Description automatically generateddisplay\_image\_task

Figure 17:display\_image\_task.

function continuously displays the current 8x8 image on an LED matrix. It initializes the ‘LedBlockDisplay ‘object and sets it up. Inside an infinite loop, it takes the mutex to safely access the shared image ‘pCurrentImage’, then calls ‘ledMatrix.display’ to show the image on the LED matrix. After displaying the image, it releases the mutex and waits for 100 milliseconds before repeating the process. This ensures that the image displayed is the latest one generated by the ‘generate\_random\_image\_task’.

### Camera Encryption Key Generator

#### State Machine

A screenshot of a computer

Description automatically generated

Figure 18: State Machine for cam\_encrypt\_gen.

#### Code

##### capture\_image(cap)

A computer screen with text

Description automatically generatedThis function grabs an image frame from the camera. It takes an instance of cv2.VideoCapture as an argument. The function tries to read a frame from the camera using cap.read(). If it succeeds, it returns the captured frame. If there's an error during the capture, it prints an error message and returns None.

Figure 19: capture\_image function.

##### preprocess\_image(frame)

A computer code with white text

Description automatically generated with medium confidenceThis function converts the captured image to a binary image. It first converts the image to grayscale using cv2.cvtColor(), and then applies a binary threshold with cv2.threshold(). Pixels above the threshold value (128) are set to white (255), and those below are set to black (0). The resulting binary image is returned.

Figure 20:preproces\_image function.

##### detect\_led\_block(binary\_image)

This function detects the LED block in the binary image. It uses cv2.findContours() to find the contours in the binary image. It then identifies the largest contour, which is assumed to be the LED block. It calculates the bounding rectangle of this contour using cv2.boundingRect(), and crops the binary image to this rectangle. The cropped image, which should contain the LED block, is then returned.

A computer code on a black background

Description automatically generated

Figure 21:detect\_led\_block function.

##### extract\_8x8\_grid(led\_block\_image)

This function extracts an 8x8 grid from the cropped LED block image. It calculates the size of each cell in the grid and initializes an 8x8 grid of zeros. For each cell, it checks the mean pixel value. If the mean value is above a certain threshold (128), the corresponding grid cell is set to 1 (which means the LED is on), otherwise it is set to 0 (which means the LED is off). The 8x8 grid is then returned.

A screen shot of a computer program

Description automatically generated

Figure 22:extract\_8x8\_grid function.

##### generate\_encryption\_key(grid)

This function generates an encryption key based on the 8x8 grid. It takes the grid and turns it into a single string, then calculates its SHA-256 hash using the hashlib.sha256() function. This generates a unique 256-bit hash, which is represented as a 64-character hexadecimal string. The resulting hash, which will be used as the encryption key, is then returned.A screen shot of a computer code

Description automatically generated

Figure 23:generate\_encryption\_key.

##### save\_images(folder, frame, binary\_image, led\_block\_image, key)

This function saves the captured image, binary image, LED block image, and encryption key to a specified folder. It first creates the folder if it does not already exist using os.makedirs(). It then saves the captured image, binary image, and LED block image as PNG files using cv2.imwrite(). The encryption key is saved to a text file.

A computer screen with text and images

Description automatically generated

Figure 24:save\_images function.

##### A screen shot of a computer program Description automatically generatedMain()

This function manages the whole process. It opens the camera using cv2.VideoCapture(0) and checks if it is working. It keeps capturing and processing images. The image is processed in several steps: preprocessing, LED block detection, grid extraction, and key generation. The captured images are displayed. The function checks for keyboard input. If the spacebar is pressed, it saves the images and key to a new folder. If 'q' is pressed, it exits the loop. The camera is released and all OpenCV windows close.

Figure 25: Main sequence function.

# Conclusion