Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 615–624
DOI: 10.15439/2025F6466
ISSN 2300-5963 ACSIS, Vol. 43

Implementation of random number generator service with IoT device

Rafał Wojszczyk, Aneta Hapka, Kacper Akdağ-Ochnik
0000-0003-4305-7253, 0000-0003-4613-2570
Koszalin University of Technology
ul. §niadeckich 2, 75-453 Koszalin, Poland
Email: {rafal.wojszczyk, aneta.hapka, u18898}@tu.koszalin.pl

***Abstract*—The paper focuses on the problem of generating random numbers, which on the surface appears to be a very simple problem. However, Most software tools used for this purpose produce pseudorandom numbers. This means that, if certain conditions are met, it is possible to reproduce successive sequences of generated numbers. This paper addresses the problem by developing an IoT device to generate true random numbers based on selected physical properties. The device communicates with a server that provides a web service and an API for generating random values in multiple variations.**

I. Introduction

IN THE DIGITAL AGE, randomness plays a crucial role in a wide array of applications, ranging from cryptography and secure communications [4] to simulations of unexpected events [3], gaming, art and GUID or UUID [5]. The quality of randomness directly impacts the performance and reliability of these applications. True Random Number Generators (TRNGs) are particularly valuable in this context, as it derive their randomness from physical processes, ensuring a higher degree of unpredictability compared to pseudo-random number generators (PRNGs), which rely on deterministic algorithms.

Randomness, as a concept, has intrigued researchers and practitioners due to its wide-ranging utility in fields such as mentioned simulations, gaming, and also statistical sampling, and experimental design. While randomness is often associated with cryptographic applications, this project focuses on non-cryptographic use cases, where high-quality random numbers are equally essential but do not require the stringent security guarantees of cryptographic systems. This distinction allows for the exploration of randomness in a broader context, emphasizing accessibility, scalability, and ease of use.

Existing solutions for generating random numbers often come with limitations, such as high costs, restricted access, or reliance on pseudo-random number generators, which, while sufficient for many applications, lack the inherent unpredictability of true randomness derived from physical processes. This paper seeks to bridge this gap by developing a scalable, cost-effective, and user-friendly web service that leverages true random number generators to deliver high-quality random numbers for non-cryptographic purposes. By focusing on applications such as simulations, Monte Carlo methods, random sampling, and artistic content generation, the service aims to provide a reliable and accessible source of randomness for a global audience, fostering innovation and experimentation across diverse domains.

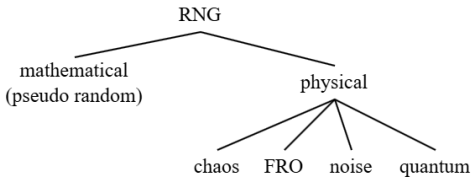

Fig. 1. Classification of random number generators.

Section II discusses how to generate random numbers. Section III describes the most important elements from the construction of the software and the device. Section IV presents and validates the system operation of the whole system and possible applications. The V section provides a summary of the work.

II. Generating random numbers

A. Classification

Random number generators can be divided into two major groups: Mathematical (pseudo random) and Physical. Pseudo random generators can be further divided into several categories based on the type of algorithm used, and physical random number generators are divided into 4 categories: noise based RNGs, free running oscillator (FRO) RNGs, chaos RNGs and quantum RNGs based on the entropy source [7].

B. Pseudo-random numbers

Pseudo-random number generators are algorithms that produce sequences of numbers approximating randomness using deterministic mathematical formulas. They rely on an initial state called a seed, which is processed into a longer sequence of values. PRNG output typically follows a uniform distribution, enabling transformations to other distributions (e.g., Gaussian) [6]. While not truly random, PRNGs are sufficient for applications like simulations, gaming, and randomized algorithms. Cryptographically secure PRNGs (CSPRNGs) enhance security by ensuring outputs are computationally indis-

tinguishable from true randomness, making them essential for cryptographic protocols [7], [8].

PRNGs excel in speed, generating millions of random numbers per second, far surpassing true random number generators limited by physical entropy sources. Their period, the length of the sequence before repetition, varies by algorithm. For instance, the Mersenne Twister has a period of $2^{19937}-1$, making it suitable for large-scale simulations, while simpler PRNGs like linear congruential generators (LCGs) have shorter periods. CSPRNGs feature periods so large that repetition is practically impossible, ensuring long-term reliability [7].

The advantages of PRNGs include low computational cost, ease of implementation, and reproducibility. By using fixed seeds, PRNGs enable consistent results for debugging and verification, making them indispensable in scientific and engineering applications [6], where repeatability is important.

There are a large number of PRNG algorithms implemented. The most popular are

1) **Middle-square method** [9]
   One of the earliest methods, presented by *J. von Neumann* in 1949 at a conference. Highly flawed, with very short period. The numbers are created by squaring the n-digit initial value, adding leading zeroes if the result has fewer than $2n$ digits and extracting middle n digits. The process can be repeated to generate more numbers.
2) **Linear Congruential Generator (LCG)** [10] One of the oldest and most widely used PRNGs, based on the Lehmer generator and published in 1958 by *W. E. Thomson* and *A. Rotenberg*. It is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m \tag{1}$$

   where:
   - X is the sequence of random values;
   - $m, 0 < m$ is the modulus;
   - $a, 0 < a < m$ is the multiplier;
   - $c, 0 \leq c < m$ is the increment;
   - $X_0, 0 \leq X_0 < m$ is the seed.

   LCGs are simple and efficient but suffer from lattice structures in higher dimensions, limiting their use in modern applications.
3) **Mersenne Twister** [11] Developed in 1997 by *Makoto Matsumoto* and *Takuji Nishimura*, this PRNG is renowned for its high-quality output and long period. Named after the Mersenne prime $2^{19937} - 1$, which defines its period length, the Mersenne Twister is widely used in simulations and statistical applications. It balances speed and statistical robustness, though its large state size can be a drawback in memory-constrained environments.
4) **Xorshift** [12] Created by *George Marsaglia*, a pioneer in random number generation, Xorshift is an extremely fast PRNG that produces sequences of $2^{32} - 1$ integers. It relies on repeated XOR operations with shifted versions of its state, making it lightweight and efficient. Variants like Xorshift+ and Xorshift* improve statistical properties while maintaining speed.
5) **WELL (Well Equidistributed Long-period Linear)** [13] An improvement over the Mersenne Twister, WELL generators offer better equidistribution and faster recovery from zero states. Developed by *F. Panneton, P. L'Ecuyer, and M. Matsumoto*, WELL variants are suitable for applications requiring high-quality randomness and long periods.
6) **PCG (Permuted Congruential Generator)** [14] A modern family of PRNGs designed by *M. E. O'Neill*, PCG combines linear congruential generators with permutation functions to enhance output quality. PCG variants are compact, fast, and statistically robust, making them popular in gaming and procedural generation.
7) **WELLDOC** [15] Aperiodic PRNGs based on infinite words techniques, developed by *L. Balkova, M. Bucci, A. de Luca, J. Hladky,* and *S. Puzynina* in 2013. These generators are theoretically interesting but less commonly used in practice due to their complexity.
8) **Blum Blum Shub** [16] A cryptographically secure PRNG based on quadratic residues modulo a product of two large primes. While highly secure, it is computationally intensive and rarely used outside cryptographic applications.
9) **Multiply-with-Carry (MWC)** [17] Developed by *George Marsaglia*, MWC generators combine multiplication and carry operations to produce long-period sequences. They are simple to implement and perform well in statistical tests.
10) **Philox and Threefry** [18] Counter-based PRNGs designed for parallel computing environments. They use cryptographic primitives to ensure high-quality randomness and are optimized for GPU and multi-core CPU architectures.

C. Physical random numbers generators

Random Number Generators that rely on physical phenomena generate inherently unpredictable and provably random outcomes. These hardware-based RNGs, often referred to as true random number generators, operate by exploiting naturally occurring random processes, such as electronic noise, quantum effects, or chaotic systems. Unlike pseudo-random number generators, which use deterministic algorithms to produce sequences that only appear random, physical RNGs are typically standalone devices connected to a computer via interfaces like USB or PCI. However, physical RNGs are not without limitations; they often exhibit bias and short-range correlations, meaning the output may not always be uniformly distributed. Bias, defined as the deviation in the probabilities of generating ones and zeros, is quantified by the equation:

$$b = \frac{p(1) - p(0)}{2} \tag{2}$$

where $p(1)$ and $p(0)$ represent the probabilities of generating a one and a zero, respectively. To mitigate these issues, post-processing techniques, such as von Neumann debiasing or

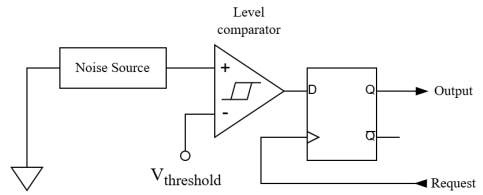

Fig. 2. The diagram shows a basic noise-based RNG comprising a noise source (e.g., thermal noise), a level comparator (with threshold $V_{threshold}$), and a D flip-flop (DFF) to sample and digitize the noise signal into random bits. This design leverages stochastic physical phenomena to generate true randomness.

cryptographic hash functions, are often applied to the raw output of physical RNGs. These methods help ensure the final output is both unbiased and suitable for applications requiring high-quality randomness, such as cryptography or secure communications.

Noise-based random number generators exploit stochastic physical phenomena to extract entropy for true randomness. These systems sample analog noise signals — unpredictable fluctuations inherent to electronic, optical, or quantum processes — and convert them into statistically independent bit sequences. Common noise sources include thermal agitation, quantum tunneling effects, and chaotic dynamics, each offering distinct advantages in entropy density and sampling rates [19], [7].

Thermal noise (Johnson-Nyquist Noise) arises from the random motion of electrons in conductive materials at equilibrium, as described by the fluctuation-dissipation theorem. Its power spectral density is given by:

$$S_V(f) = 4k_BTR \quad (\text{V}^2/\text{Hz}), \tag{3}$$

where k_B is Boltzmann's constant, T is the absolute temperature (K), and R is the resistance (Ω). This white noise exhibits a Gaussian amplitude distribution, making it a robust entropy source. Practical implementations sample thermal noise via high-gain amplifiers and analog-to-digital converters (ADCs), though challenges include mitigating $1/f$ noise at low frequencies and temperature-induced drift [7].

Zener noise (Avalanche Breakdown Noise) originates from quantum tunneling and impact ionization in reverse-biased Zener diodes operating near the avalanche breakdown regime. Individual electron-hole pair generation events produce discrete current spikes, yielding a shot noise process with Poissonian statistics. The noise current I_n follows:

$$\langle I_n^2 \rangle = 2qI_{\text{DC}}\Delta f, \tag{4}$$

where q is the electron charge, I_{DC} is the bias current, and Δf is the bandwidth. Zener-based RNGs achieve high bitrates (>100 Mbps) but require precise voltage regulation to stabilize the breakdown region and avoid deterministic oscillations [7].

Chaos-based random number generators leverage nonlinear dynamical systems governed by chaos theory to harvest entropy. These systems exhibit deterministic yet unpredictable behavior due to their sensitivity to initial conditions (the "butterfly effect") and topological mixing. Common implementations exploit chaotic phenomena in optical, electronic, optoelectronic, or mechanical domains, where small perturbations amplify exponentially over time, yielding outputs that are statistically indistinguishable from true randomness [21]. Other example of optical chaos-based RNGs employs distributed feedback (DFB) lasers with self-feedback mechanisms.

Free-Running Oscillators [7] (FROs) are a class of hardware-based random number generators that exploit the inherent jitter and metastability in electronic circuits to produce randomness. An FRO is a simple yet effective circuit that consists of a logical inverter whose output is fed back into its input, creating an unstable oscillatory state. The oscillation frequency is determined by the internal propagation delays of the inverter and the stray capacitances in the circuit. This unpredictability makes FROs a popular choice for generating random numbers in embedded systems and hardware security applications. The randomness in an FRO-based RNG is extracted by sampling the oscillating signal at irregular intervals or by comparing the phases of multiple FROs. One common approach is to use a pair of FROs with slightly different frequencies. The phase difference between the two oscillators drifts over time due to jitter, and this drift is sampled to produce random bits.

Quantum Random Number Generators (QRNGs) [8], [20] leverage the inherent unpredictability of quantum mechanics to generate true randomness. Unlike classical random number generators, which rely on deterministic algorithms or physical processes that may be predictable in principle, QRNGs exploit the fundamental indeterminism of quantum systems. This makes them ideal for applications requiring high levels of security, such as cryptography and secure communications. QRNGs operate by measuring quantum phenomena that are intrinsically random. One of the most common implementations uses photons, the elementary particles of light, and their quantum properties. For example, a photon in a superposition of states will collapse randomly into one of two possible states when measured. This randomness is used to generate binary bits (0s and 1s) [7], [8].

Spatial QRNGs [8], [7] rely on the physical path a photon takes after passing through a beam splitter. These systems use two detectors, one for each output port of the beam splitter. While conceptually simple, spatial QRNGs require careful calibration to ensure that the detectors are equally sensitive and that the beam splitter is perfectly aligned. Any imbalance in the system can introduce bias into the output, requiring post-processing to correct.

Temporal QRNGs [8], [7], on the other hand, exploit the randomness in the timing of photon arrivals. Instead of measuring the path a photon takes, these systems measure the time intervals between successive photon detections. For example, the system might compare two consecutive time intervals. This approach has the advantage of requiring only a single detector, simplifying the hardware design and reducing calibration requirements.

Another class of QRNGs exploits the randomness of radioactive decay [8]. In these systems, a radioactive source emits particles (e.g., alpha or beta particles) at random intervals. A detector registers the arrival of these particles, and the timing of the detections is used to generate random bits.

D. TRNG Type Comparison

The development and evaluation of TRNGs have been extensively studied in the literature. Quantum-based TRNGs, in particular, have gained significant attention due to their provable security based on the laws of quantum mechanics. Herrero-Collantes and Garcia-Escartin [8] provide a comprehensive review of quantum random number generators (QRNGs), highlighting their theoretical foundations and practical implementations. They emphasize the role of quantum phenomena, such as photon detection and vacuum fluctuations, in generating true randomness.

Cirauqui et al. [20] further explore the challenges and benchmarking of QRNGs, discussing the trade-offs between speed, security, and resource requirements. Their work underscores the importance of device-independent QRNGs, which offer enhanced security by minimizing assumptions about the underlying hardware.

Stipčević and Koç [7] provide a broader perspective on TRNGs, covering chaos-based, noise-based, and FRO-based designs. They discuss the sources of entropy, post-processing techniques, and security considerations for each type. Their work highlights the importance of environmental robustness and cost-effectiveness in practical TRNG implementations.

True Random Number Generators vary in design and application. Chaos-based TRNGs use nonlinear systems for high entropy but require complex calibration, making them suitable for cryptography. FRO-based TRNGs, leveraging electronic jitter, are simple and fast but require post-processing, which is ideal for IoT and FPGAs. Noise-based TRNGs rely on thermal or shot noise, offering reliability but requiring analog components, often used in hardware security modules (HMS) and military systems. Quantum-based TRNGs, exploiting quantum phenomena, provide ultimate security but are costly and sensitive, limiting their use to high-security applications. Each TRNG type balances performance, cost, and complexity for specific needs.

The choice of TRNG type depends on the specific application requirements:

- Chaos-based TRNGs are well-suited for cryptographic applications due to their high entropy and nonlinear dynamics. However, they are sensitive to environmental conditions and require careful calibration.
- FRO-based TRNGs are ideal for resource-constrained environments, such as IoT devices and FPGAs, due to their simplicity and low power consumption. However, they often require post-processing to improve randomness quality.
- Noise-based TRNGs offer a balance between reliability and security, making them suitable for hardware security modules and military systems. Their reliance on analog components increases the complexity of the design.
- Quantum-based TRNGs provide unparalleled security, making them ideal for high-stakes applications such as quantum communication and cryptography. However, their high cost and sensitivity to environmental factors limit their widespread adoption.

III. Implementation

A. Motivation

Randomness is seemingly underestimated, but it plays a key role in many areas: from simulations and Monte Carlo methods, through computer games, to statistical sampling and the generation of unique identifiers. The quality of random numbers directly affects the reliability of calculation results and system security, which is why generators based on real physical processes, rather than solely on deterministic algorithms, are increasingly being used. There is no common belief among programmers that it is necessary to use truly random generators. Popular libraries offer good-quality pseudo-random generators, but their deterministic nature makes it possible to reproduce the sequence if the initial state is known. True random generators, using the previously described thermal noise, quantum phenomena, or oscillator jitter, provide a higher degree of unpredictability. Building your own TRNG allows you to eliminate dependence on external services and gives you full control over the quality and availability of the entropy source.

The implementation of the true random number generator tool described below is a response to the needs of programmers. The basic access layer for this type of user is WebAPI, hence a lot of attention was paid to the preparation of the API and documentation when developing the solution. On the other hand, it is valuable to be able to test the solution in advance without having to use typical developer tools. In order to meet this requirement, a web interface was developed.

B. System architecture

The system must accommodate a moderate volume of simultaneous requests while maintaining low latency, high reliability, and scalability, ensuring robust performance under variable operational conditions. Additionally, the implementation will incorporate cryptographic protocols to secure data transmission and prevent adversarial manipulation during distribution.

Components presented in Fig. 3 work together in order to generate random numbers and serve them on the internet. The web app acts as a graphical interface for users. It displays the services available on the website as interactive forms that users use to send requests to the API and presents the results - the generated random numbers. The Hardware Random Number Generator is the source of random numbers. It generates random 32 bit words as quickly as possible and sends them over a WebSocket connection to the web API. The web API in pair with the database performs the back-end duties: converting random bits into numbers, processing user

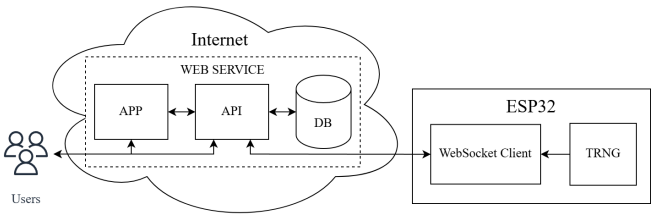

Fig. 3. The web service is a containerized application deployed in the cloud. The TRNG is a physical module connected to the internet. Users access the app through the internet.

requests, handling authorization, authentication and sending the results to users. The app, API and database components are deployed in a containerized environment in the cloud.

The development of the system utilizes a diverse set of tools and technologies tailored to each component. For the Web App, HTML structures web content, CSS handles styling, and JavaScript adds interactivity, while JQuery simplifies DOM manipulation and Bootstrap ensures responsive design. The Web API relies on Python and Django, with Django Rest Framework enabling RESTful API development, Open API providing standardization, and ASGI with Daphne supporting asynchronous communication. HTTP and WebSocket protocols facilitate seamless data exchange between the server and clients. The Database uses SQLite for its lightweight, high-reliability SQL engine, with SQL managing data manipulation and retrieval. For the TRNG, it use an IoT device [2], it means the ESP32 microcontroller with an embedded TRNG is programmed using the Arduino IDE and C++, ensuring efficient and secure random number generation. Together, these tools and technologies provide a robust foundation for the system's development. The ESP32 microcontroller was chosen because of its built-in Wi-Fi module (which simplified communication implementation), low purchase cost, and ease of programming in the Arduino environment. The IoT device can also be built on other microcontrollers that support Arduino programming, such as those from the AVR and NRF families.

C. Software

The web application's front end, which serves as the graphical interface accessible via a web browser, is integrated into the Python Django application using the Django templating engine. While the front end is primarily written using HTML, CSS, JavaScript and the Django Template Language (DTL), the Django templating engine dynamically renders the pages based on the context provided by the backend. This integration allows the front end to seamlessly interact with the Django framework.

The Web API is a distributed system designed to provide secure, scalable, and efficient access to random number generation services. It is composed of three interconnected applications, each responsible for a distinct functional domain. These applications work in concert to handle user authentication, random number generation and data delivery, ensuring a robust, modular architecture.

The Web API is organized into the following components:

1) **API (Application Programming Interface):** The core application responsible for handling user requests and delivering random number generation services. It implements the following modules:
   - **Views and Models:** Defines the endpoints and data structures for random tools, such as integer sequences, floating-point numbers, and custom distributions.
   - **Random Tool Functions:** Implements algorithms for transforming raw random bits into user-requested formats.
   - **OpenAPI Documentation:** Provides a standardized, machine readable specification of the API endpoints, request/response formats, and authentication mechanisms. This documentation is automatically generated using Redoc, ensuring clarity and consistency for developers integrating with the API. Available at https://trng.kacperochnik.eu/api/
2) **Auth (Authorization and Authentication Module):** A dedicated service for managing user identities, access control, and resource quotas. Key features include:
   - **User Authentication:** Implements Session and JWT-based authentication to verify user credentials and issue access tokens.
   - **Authorization:** Enforces role-based access control (RBAC) to restrict access to sensitive endpoints (e.g., administrative tools).
   - **Points System:** Tracks and deducts user points for each random number generation request, ensuring fair usage and preventing abuse.
3) **Trng (True Random Number Generator Interface):** A WebSocket-based service that interfaces with physical TRNG hardware to harvest entropy and deliver it to the API. Its components include:
   - **WebSocket Consumer:** Manages real-time connections with TRNG devices, enabling low-latency data transfer.
   - **Circular Buffer:** Stores incoming random bits in a fixed-size, ring-shaped buffer to ensure continuous availability and prevent data loss during high-throughput operations.
   - **Bit Disposal Interface:** Provides a standardized interface for delivering random bits to the API's random tool functions, ensuring compatibility with diverse TRNG hardware.

The system employs SQLite as its relational database management system (RDBMS), leveraging Django's Object-Relational Mapping (ORM) layer for data persistence operations. SQLite was selected for its minimal configuration requirements, serverless architecture, and atomic transaction support - characteristics that align with the application's need for simplicity and rapid development iteration.

D. Hardware

The Hardware Random Number Generator (HRNG) subsystem comprises an ESP32-WROOM-32D microcontroller configured as a dedicated entropy source. This IoT-enabled device leverages the chip's native True Random Number Generator peripheral to harvest physical noise, transmitting raw random data to the web API via secure WebSocket protocol. The complete design files and firmware are available in the project repository.[1]

The ESP32 incorporates a True Random Number Generator that generates 32-bit random numbers suitable for cryptographic operations [23]. The TRNG relies on physical entropy sources so that no number within the specified range is more or less likely to appear than any other. In terms of key characteristics, the generator obtains its entropy from thermal noise produced by the high-speed ADC and SAR ADC, as well as from an asynchronous clock mismatch provided by the RC_FAST_CLK (8 MHz internal RC oscillator). Its output rate can theoretically reach up to 5.2 Mbps, however, to ensure optimal performance, the recommended read rate is 500 kHz when using SAR ADC noise and 5 MHz when using high-speed ADC noise. The TRNG is accessed via the 'RNG_DATA_REG' memory-mapped I/O (MMIO) register at the address '0x3FF75144', and it incorporates built-in Von Neumann debiasing with a 2:1 bit compression ratio to remove bias from the raw entropy stream.

Functionally, the TRNG generates true random numbers based on two primary sources of entropy. The first source, Thermal Noise, is generated by both the high-speed ADC and the SAR ADC, when enabled, these ADCs produce bit streams that serve as random seeds for the TRNG. The second source, Asynchronous Clock Mismatch, is derived from the RC_FAST_CLK (8 MHz internal RC oscillator), which introduces timing jitter and clock drift. These sources are combined using XOR logic, as expressed in the following equation:

$$b_{out} = \bigoplus_{i=0}^{n} (s_i \oplus t_i) \tag{5}$$

where:

- s_i represents sampled oscillator states (e.g., from the SAR ADC or high-speed ADC).
- t_i represents timing jitter measurements (e.g., from the RC_FAST_CLK).

Regarding the entropy feed mechanism, the TRNG uses two distinct noise sources. SAR ADC Noise provides 2 bits of entropy per clock cycle of the RC_FAST_CLK (8 MHz), with a maximum recommended read rate of 500 kHz to ensure maximum entropy. In contrast, High-Speed ADC Noise offers 2 bits of entropy per APB clock cycle (typically 80 MHz) and has a maximum recommended read rate of 5 MHz to similarly ensure optimal entropy extraction.

The Arduino program is fairly simple, it sends generated 32 bit random words, over WebSocket, as fast as possible, in a loop. It uses esp32 random API function - *esp_random()* to retrieve random bits from the register. Source code are presented in example 3.1.

[1]Repository: https://github.com/TeriyakiGod/esp-arduino-trng

Example 3.1 (C++):

```
void loop() {
  webSocket.loop();// Maintain WebSocket conn.
  // Check if WiFi is still connected
  if (WiFi.status() != WL_CONNECTED) {
    Serial.println("WiFi lost connection");
    connectToWiFi();
  }
  // Gen. rnd. number and send over websocket
  uint32_t randomValue = esp_random();
  webSocket.sendBIN((uint8_t*)&randomValue(),
  sizeof(randomValue));
}
```

E. Security

The developed solution belongs to the class of IoT devices and is constantly connected to the Internet, so adequate security is required [2]. The Web API incorporates several integrated mechanisms to ensure both security and scalability. It encrypts all communications between clients and the API using TLS 1.3, which protects against eavesdropping and man-in-the-middle attacks. In addition, the API employs rate limiting with token bucket algorithms to throttle excessive requests, thereby preventing denial-of-service (DoS) [4] attacks and ensuring fair distribution of resources. Load balancing is also applied, as incoming requests are distributed across multiple API instances via round-robin or least-connections strategies to maintain high availability and fault tolerance. Complementing these measures, robust authentication is achieved through the use of session cookies (maintained via a 'sessionid') and API tokens supplied in the 'Authorization' header, while standard error responses—such as '403 Forbidden' for insufficient points or privileges, '503 Service Unavailable' when the TRNG is offline, and '429 Too Many Requests' when the rate limit is exceeded—provide clear feedback to users.

The device further implements a custom binary protocol over WebSocket Secure (WSS) that is optimized for both efficiency and security. This protocol defines its frame format using 32-bit little-endian unsigned integers and supports a transmission rate of around 20 KB/s. Error handling is addressed by incorporating an exponential backoff strategy for connection failures, and security is reinforced by using TLS 1.3 in conjunction with the ESP32's root certificate bundle. Moreover, additional security considerations include physical isolation through a Faraday-shielded enclosure, runtime protection ensured by secure boot with flash encryption, and enhanced transport security via Perfect Forward Secrecy (PFS) using ECDHE-RSA. Together, these strategies result in an implementation that achieves 128-bit security under the Dolev-Yao threat model, provided that there is no physical access to the TRNG unit.

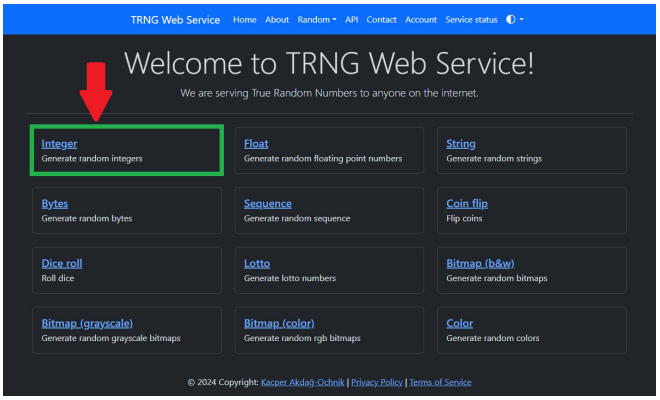

Fig. 4. Homepage with the Integer Generator button highlighted

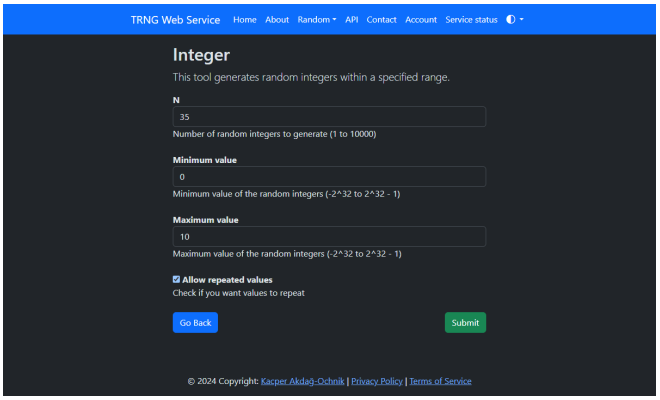

Fig. 5. Integer generator form filled with values.

IV. Verification

A. Access via GUI and API

Users begin by visiting the web service through a browser, where the homepage presents a selection of random number generators. To generate random integers, the user navigates to the integer generator using a button on the homepage, as shown in Fig. 4. Once on the integer generator page, the user can configure the tool to their preferences. For example, to generate 35 integers in the range from 0 to 10, the user sets N (the number of random integers) to 35, the minimum value to 0, and the maximum value to 10, as illustrated in Fig. 5. After submitting the form, the application returns the generated random integers. The distribution of these integers is visualized in a histogram in Fig. 6, showing a roughly uniform spread of random values across the specified range.

For programmatic access, users can send a GET request to the API endpoint[2]. The API responds with a JSON object containing the generated random integers, a timestamp, and the number of bits used, as shown in the code listing 4.1. The histogram in Fig. 7 visualizes the distribution of the integers generated via the API.

[2]https://serveraddess/api/rand/int?n=35&min=0&max=10&repeat=True

Example 4.1 (JSON):


```
"values":
    [4, 8, 2, 9, 8, 9, 10, 10, 5, 3, 1, 0,
    10, 6, 10, 3, 6, 10, 7, 1, 10, 3, 4,
    2, 10, 6, 8, 5, 3, 0, 9, 0, 0, 6, 7],
"timestamp": "2025-03-20T17:14:12.492409",
"bits": 1120
```


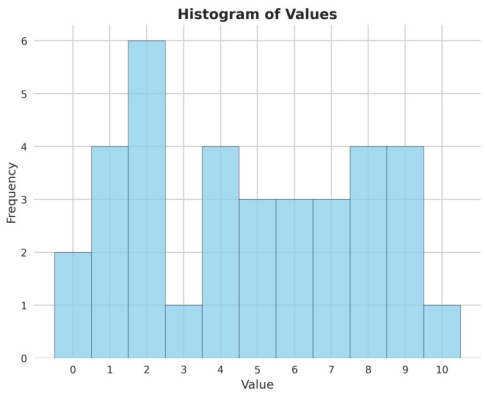

Fig. 6. Histogram of the first result

The application provides a seamless experience for generating random numbers, whether through the intuitive UI or the programmatic API. The histograms in Figs. 6 and 7 demonstrate that the generated integers are roughly uniformly distributed and random each time, validating the effectiveness and reliability of the random number generator. This ensures that users can trust the application to produce unbiased and balanced results for their needs. All functions are listed in the V appendix section.

B. Statistical Testing with Dieharder

The randomness of the TRNG output was evaluated using the Dieharder test suite, a comprehensive collection of statistical tests designed to assess the quality of random number generators. Dieharder analyzes the output for patterns,

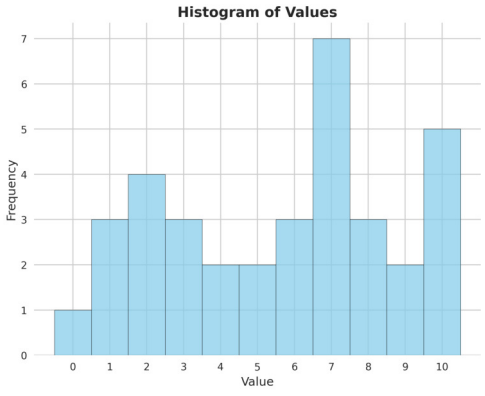

Fig. 7. Histogram of the second result

TABLE I
DIEHARDER TEST RESULTS. ALL OF THE TESTS PASSED, ONLY 2 WEAK RESULTS: RGB_BITDIST AND RGB_LAGGED_SUM.

Test Name	ntup	tsamples	psample	p-value
diehard_birthdays	0	100	100	0.01192945
diehard_operm5	0	1000000	100	0.67160533
diehard_rank_32x32	0	40000	100	0.88503575
diehard_rank_6x8	0	100000	100	0.44944912
diehard_bitstream	0	2097152	100	0.07796698
diehard_opso	0	2097152	100	0.75986760
diehard_oqso	0	2097152	100	0.52389989
diehard_dna	0	2097152	100	0.63689771
diehard_count_1s_str	0	256000	100	0.64745948
diehard_count_1s_byt	0	256000	100	0.48743869
diehard_parking_lot	0	12000	100	0.06943192
diehard_2dsphere	2	8000	100	0.33036560
diehard_3dsphere	3	4000	100	0.69729265
diehard_squeeze	0	100000	100	0.41006273
diehard_sums	0	100	100	0.22268815
diehard_runs	0	100000	100	0.87899505
diehard_craps	0	200000	100	0.01235318
marsaglia_tsang_gcd	0	10000000	100	0.47170575
sts_monobit	1	100000	100	0.72827629
sts_runs	2	100000	100	0.52336996
sts_serial	1	100000	100	0.03008230
rgb_bitdist	6	100000	100	0.00048266
rgb_min_distance	2	10000	1000	0.16471595
rgb_permutations	2	100000	100	0.26856586
rgb_lagged_sum	24	1000000	100	0.99977196
rgb_kstest_test	0	10000	1000	0.27646420
dab_bytedistrib	0	51200000	1	0.52014772
dab_dct	256	50000	1	0.83006246
dab_filltree	32	15000000	1	0.79259371
dab_monobit2	12	65000000	1	0.12926929

biases, and correlations that could indicate deviations from true randomness. The TRNG output was subjected to a variety of tests, and the results are presented in table I.

The Dieharder test results confirm that the TRNG output behaves as expected for a high-quality random sequence. Most tests passed with p-values within acceptable confidence intervals, indicating no significant biases or correlations. However, a few tests, such as rgb_bitdist (for bit distances 4 and 6) and rgb_lagged_sum (for lag 24), returned weak results. The weak result of the two tests is caused by insufficient variability in the sequences of consecutively generated zeros or ones. This may be due to suboptimal threshold alignment in the thermal noise signal digitization circuit, leading to correlation between consecutive samples.

C. Possible applications

The proposed self-hosted true random number generation system addresses diverse applications requiring high-quality, statistically robust, and verifiably unbiased randomness. The scientific field can make use of the web service in many ways, eg. Monte Carlo Simulations, Diffusion Models and Statistical Research.

Large-scale computational models in fields such as quantum chemistry, particle physics, and financial mathematics rely on entropy-rich seeds to minimize correlation artifacts. For instance, lattice Boltzmann simulations of fluid dynamics and option pricing models in quantitative finance demand randomness with certified uniformity to ensure convergence accuracy. The system's post-processing algorithms, which suppress residual biases in raw entropy sources, enhance the reproducibility of results across distributed computational clusters.

Modern generative artificial intelligence frameworks, including stochastic differential equation (SDE)-based diffusion models, utilize random noise sequences to synthesize high-fidelity images or molecular structures. A TRNG-backed service guarantees that the latent space sampling process remains free from deterministic patterns, which could otherwise introduce perceptual biases or reduce generative diversity in outputs.

Randomized controlled trials (RCTs), bootstrap sampling, and permutation tests require auditable randomness to preserve methodological integrity. By providing cryptographically signed random streams, the system enables researchers to verify the provenance of experimental data, mitigating concerns about inadvertent selection biases or adversarial tampering in peer-reviewed studies.

There are multiple applications in the field of gaming.

1) **Multiplayer Games:** Massively multiplayer online (MMO) platforms and blockchain-based play-to-earn ecosystems necessitate transparent randomness for procedural content generation, loot box mechanics, and non-fungible token (NFT) attribute assignment. Centralized pseudorandom number generators often face skepticism from users; outsourcing to an independently audited TRNG service enhances player trust and regulatory compliance.
2) **Lotteries and Raffles:** Public or decentralized systems conducting randomized prize allocations require algorithmic transparency to demonstrate impartiality. Integrating the TRNG system with blockchain-based smart contracts or verifiable computation frameworks ensures that draw outcomes derive from non-deterministic physical entropy, enabling participants to independently audit the randomness generation process. This eliminates skepticism toward algorithmic fairness, particularly in decentralized or distributed systems where trust in centralized authorities is limited.
3) **Casino Gaming:** Digital gaming platforms depend on unpredictability for mechanics such as shuffling virtual decks, generating roulette ball trajectories, or determining slot machine payouts. Hardware-backed TRNGs mitigate vulnerabilities inherent to algorithmic pseudorandomness—such as state reconstruction or seed manipulation—by anchoring outcomes to physically irreproducible entropy sources. This ensures that game mechanics cannot be reverse-engineered or biased, even by privileged insiders with access to the software stack.

The web service can be applied to create art.

1) **Generative Art:** Algorithmic art platforms (e.g., Art Blocks) and fractal geometry engines depend on entropy to drive procedural variations. High-dimensional randomness ensures that procedurally generated artworks—such as those minted as NFTs—exhibit unique traits without deterministic repetition, thereby preserving artistic value and collector interest.
2) **Algorithmic Music:** Stochastic composition tools, such as Markov chain-based melody generators or granular synthesis systems, leverage random parameters to explore unconventional harmonic structures. TRNG-derived sequences facilitate organic, non-repetitive soundscapes, distinguishing outputs from those produced by conventional PRNGs with limited cycle lengths.
3) **Interactive Installations:** Immersive environments and augmented reality (AR) experiences use real-time randomness to adapt narrative pathways or visual elements based on user interactions. A low-latency TRNG service enables dynamic, audience-responsive installations while maintaining statistical unpredictability across prolonged exhibitions.

While the system is unsuitable for high-throughput cryptographic operations like TLS handshakes (handling a single request takes almost 380 ms, for example from section IVa), it may supplement key derivation for non-real-time applications, such as creating master keys for offline data vaults and random vectors [1]. However, dedicated hardware security modules remain preferable for latency-sensitive tasks.

However, in production use, the scalability issue can be solved by duplicating instances through containerization and implementing a load balancer. Compared to competing solutions, it is worth highlighting the open-source approach and documentation of all endpoints. In addition, the proposed solution meets functional security requirements, including TLS and JWC.

V. Summary

Generating true random numbers is often underestimated by programmers. However, it turns out that a great deal depends on it, and it should be as random as possible. The paper demonstrates a software and hardware solution for generating such numbers. The IoT device developed uses noise, which serves as the basis for the generation. The generator's functions are then accessed via an API and GUI on a website.

The randomness of the generated numbers was verified using automatic tests, manual tests (histograms presented earlier) and a dedicated randomness verification solution, Dieharder. The results obtained show that the whole in the sense as a system generates data with a high level of randomness. Functional tests confirm that the system as a whole generates and provides numbers in accordance with user requirements and handles errors correctly. Performance tests show that the selected architecture (cloud containers, WebSocket, rate limiting) ensures low latency and high scalability. Statistical tests show that bit strings from TRNG do not exhibit significant deviations from randomness: no bias, appropriate distribution, no autocorrelation. As a result, they can be considered "truly random."

Obtaining a high level of randomness makes it possible to excavate the solution in many areas of IT. However, it should be pointed that the speed of data generation is average (less than 3 requests per second), so it cannot be used in real time. The solution to this problem is to multiply the IoT device, which, thanks to the use of ESP32, will not be expensive. Other risks arise from the analog circuit and the analog-to-digital converter. There is a risk that after generating a strong and steady signal, the system will receive the same grain all the time, hence work on improving the analog part is being considered. Other developments are related to expanding the software layer to generate random data for sharpening in artificial intelligence tools. Future work also could involve testing larger datasets to ensure the TRNG's performance scales appropriately. Additionally, other statistical test suites, such as the NIST Statistical Test Suite [22] or TestU01, could be used to further validate the TRNG's randomness.

Appendix

The system performs following functions:

1) Generating random numbers:
   a) Generating n random integers in range $[a, b]$.
   b) Generating n random floating point numbers in range $[0, 1]$ with p precision.
   c) Generating n random bytes represented in selected positional numeral system:
      i) Binary
      ii) Octal
      iii) Hexadecimal
      iv) Decimal
   d) Generating n random strings m long, with options:
      i) With repeating characters.
      ii) Without repeating characters.
      iii) With character set selection:
         A) Letters
         B) Digits
         C) Special characters
   e) Generating a random permutation of integers in range $[a, b]$
   f) Generating n coin tosses.
   g) Generating n, m sided dice throws.
   h) Generating n lottery tickets.
   i) Generating a random bitmap with dimensions x by y in PNG format, with the following types:
      i) Monochrome (1 bit per pixel)
      ii) Grayscale (8 bits per pixel)
      iii) Color (24 bits per pixel)
   j) Generating n random HEX color codes.
2) Ability to communicate with True Random Number Generator Modules using WebSocket protocol.

3) Ability to exchange data with users using HTTP protocol.
4) Encrypted connections using SSL/TLS.
5) Access to the service through the Web App or Web API.
6) Access control.
7) Anonymous user access with IP registry.
8) Web API keys for registered users.
9) Credit system allowing users to exchange points for random numbers.

REFERENCES

[1] Rak, Tomasz, and Dariusz Rzońca. 2024. "Security and Privacy in Networks and Multimedia" Electronics 13, no. 15: 2887. doi: https://doi.org/10.3390/electronics13152887

[2] Felkner, Anna, Jan Adamski, Jakub Koman, Marcin Rytel, Marek Janiszewski, Piotr Lewandowski, Rafał Pachnia, and Wojciech Nowakowski. 2024. "Vulnerability and Attack Repository for IoT: Addressing Challenges and Opportunities in Internet of Things Vulnerability Databases" Applied Sciences 14, no. 22: 10513. doi: https://doi.org/10.3390/app142210513

[3] Łukasz Sobaszek and Arkadiusz Gola and Edward Kozłowski, Application of survival function in robust scheduling of production jobs, editors M. Ganzha and L. Maciaszek and M. Paprzycki, Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, vol. 11, 2017, doi: 10.15439/2017F276

[4] Daniel Czyczyn-Egird, Rafał Wojszczyk, "The effectiveness of data mining techniques in the detection of DDoS attacks". In: Omatu, S., Rodríguez, S., Villarrubia, G., Faria, P., Sitek, P., Prieto, J. (eds) Distributed Computing and Artificial Intelligence, 14th International Conference. DCAI 2017. Advances in Intelligent Systems and Computing, vol 620. Springer, Cham. doi: https://doi.org/10.1007/978-3-319-62410-5_7

[5] T. Nowicki, K. Chlebicki, D. Pierzchała, R. Waszkowski and K. Worwa, "Simulation method of reliability evaluation for RFID based restricted access administrative office," 2017 IEEE International Conference on RFID Technology & Application (RFID-TA), Warsaw, Poland, 2017, pp. 89-94, doi: 10.1109/RFID-TA.2017.8098877.

[6] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. Automatic nonuniform random variate generation. Springer Science & Business Media, 2013.

[7] Mario Stipčević and Çetin Koç. "True Random Number Generators". In: Nov. 2014, pp. 275–315. isbn: 978-3-319-10682-3. doi: 10.1007/978-3-31910683-0_12.

[8] Miguel Herrero-Collantes and Juan Carlos Garcia-Escartin. "Quantum random number generators". In: Reviews of Modern Physics 89.1 (Feb. 2017). issn: 1539-0756. doi: 10.1103/revmodphys.89.015004. url: http://dx.doi.org/10.1103/RevModPhys.89.015004

[9] John von Neumann. "Various techniques used in connection with random digits". In: Monte Carlo Method. Ed. by A. S. Householder, G. E. Forsythe, and H. H. Germond. Vol. 12. National Bureau of Standards Applied Mathematics Series. Washington, D.C.: U.S. Government Printing Office, 1951, pp. 36–38

[10] Derrick H. Lehmer. "Mathematical methods in large-scale computing units". In: Proceedings of 2nd Symposium on Large-Scale Digital Calculating Machinery. 1951, pp. 141–146.

[11] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: ACM Trans. Model. Comput. Simul. 8.1 (Jan. 1998), pp. 3–30. issn: 1049-3301. doi: 10.1145/272991.272995..

[12] George Marsaglia. "Xorshift RNGs". In: Journal of Statistical Software 8.14 (2003), pp. 1–6. doi: 10.18637/jss.v008.i14.

[13] François O Panneton, Pierre l'Ecuyer, and Makoto Matsumoto. "Improved long-period generators based on linear recurrences modulo 2". In: ACM Transactions on Mathematical Software 32.1 (2006), pp. 1–16. doi: 10.1145/1132973.1132974.

[14] Melissa E. O'Neill. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Technical Report HMCCS-2014-0905. Harvey Mudd College, Sept. 2014.

[15] L'ubomíra Balková et al. "Aperiodic pseudorandom number generators based on infinite words". In: Theoretical Computer Science 647 (Sept. 2016), pp. 85 - 100. issn: 0304-3975. doi: 10.1016/j.tcs.2016.07.042.

[16] Lenore Blum, Manuel Blum, and Michael Shub. "A Simple Unpredictable Pseudo-Random Number Generator". In: SIAM Journal on Computing 15.2 (1986), pp. 364–383. doi: 10.1137/0215025.

[17] George Marsaglia. "Random number generators". In: Journal of Modern Applied Statistical Methods 2.1 (May 2003), pp. 2–13. doi: 10.22237/jmasm/1051747320.

[18] John Salmon et al. "Parallel random numbers: as easy as 1, 2, 3". In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. 2011, Article No. 16. doi: 10.1145/2063384.2063405.

[19] C. D. Motchenbacher and J. A. Connelly. Low-noise electronic system design. Wiley Interscience, 1993. isbn: 0-471-57742-1.

[20] David Cirauqui and Miguel Ángel García-March and Guillem Guigó Corominas and Tobias Graß and Przemysław R. Grzybowski and Gorka Muñoz-Gil and J. R. M. Saavedra and Maciej Lewenstein. Quantum Random Number Generators : Benchmarking and Challenges. 2022.

[21] X. Li et al. "Scalable parallel physical random number generator based on a superluminescent LED". In: Opt. Lett. 36 (2011), pp. 1020–1022.

[22] Darren Hurley-Smith, Julio Hernandez-CastroJ, ulio Hernandez-Castro, Bias in the TRNG of the Mifare DESFire EV1, RFIDSec 2016

[23] Espressif Systems. ESP32 Technical Reference Manual. Accessed: 2023-10-01. 2023. url: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf\#rng.