Exploring Digital Pseudorandom Number Generation

1st Shane Gordon
Computer Science: Robotics and Intelligent Systems
Colorado School of Mines
Golden, USA
shanemgordon@mines.edu

2nd Will Ramsey
Electrical Engineering and Computer Science
Colorado School of Mines
Golden, USA
wramsey@mines.edu

3rd Nicholas Iverson
Computer Science
Colorado School of Mines
Golden, USA
niverson@mines.edu

4th Roberto Garcia
Computer Science
Colorado School of Mines
Golden, USA
rgarcia1@mines.edu

I. Introduction

Pseudorandom number generators (PRNGs) are core components of modern digital and computing systems. They provide the foundation for secure information transmission and communication, data integrity over this communication, and also in simulation fidelity - simulating real world phenomenon that can't be predicted. The numbers that PRNGs generate aren't truly random, but PRNGs aim to approximate randomness through a deterministic process, which typically involves seed-based algorithms that produce long statistically uniform sequences. In cryptographic applications where the quality of randomness affects system security, these PRNGs must withstand statistical prediction, state reconstruction, and low entropy.

In the following project, we investigate the design of a low-cost, accessible for this exact application. This project also aims to learn about existing solutions, the process of testing a virtual PRNG pipeline that sources entropy from various accessible API's that can be found on the Internet from anywhere in the world. The API's that we investigated involved data that had no statistical relation with historic data that could be accessed, which decreased the predictability of the outcome, directly increasing the entropy. These API's included sports statistics, large language model outputs, cosmic ray detections, and crime data. Our goal is to analyze whether such sources can be combined, post-processed, and utilized to produce high-quality pseudorandom output suitable for secure, practical applications.

A. Uses of PRNGS

PRNGs are in every facet of computing where repeatable randomness is required. In operating systems, PRNGs are the core security mechanism. For example, Address Space Layout Randomization randomizes the memory addresses where code segments, the stack, and heap are loaded. This makes high priority and confidential program or pieces of data in an unknown part of hardware to an attacker, forcing them to have to guess the memory locations. Without high-quality randomness ASLR is predictable and allows attackers to execute return-oriented programming or buffer overflow attacks.

In secure communication, there must exist protocols for machines to receive and send information to the next machine. When there are common knowledge protocols in existence when dealing with confidential information, it is crucial to have defense mechanisms for attackers who know how to exploit these protocols. Just as cash-in-transit vehicles are equipped with heavy armor and armed guards when delivering money to a bank, secure communication needs protection over the route and the protocol for taking data from the wire and into the memory of a machine. This is where PRNGs are critical. PRNGs generate ephemeral session keys and cryptographic nonces—values that must be unique and unpredictable for each session. If these values are reused or predictable, it can compromise the confidentiality of the entire encrypted session. For instance, predictable nonces in AES-GCM can lead to complete key recovery due to IV reuse vulnerabilities.

In a less consequential example, procedural generation in games such as the creation of randomized terrain, item drops, and AI behavior also relies on PRNGs. These systems often use seeded generators to allow for consistent world-building from a single input seed. This allows games like Minecraft to generate massive, complex environments with minimal storage, using only the seed and a deterministic algorithm.

Cryptographic systems represent the most consequential use cases. A cryptographically secure PRNG (CSPRNG) must meet far stricter criteria than conventional PRNGs. A notable historical failure is the Debian OpenSSL vulnerability (CVE-2008-0166), where the entropy used for seeding OpenSSL's PRNG was reduced to the process ID due to a patch that silenced uninitialized memory warnings. This led to only

32,000 possible keys being generated across millions of systems, allowing for precomputation attacks and mass compromise of keys [8]. *How does a secure PRNG work?* CSPRNGs maintain an internal state, seeded with high-entropy input. They use deterministic cryptographic algorithms to expand this entropy into long sequences of output. These algorithms often employ AES in counter mode, HMAC constructions, or cryptographic hash functions such as SHA-3 to prevent correlation between output bits and internal state. To maintain forward and backward secrecy, CSPRNGs must support reseeding — adding fresh entropy from trusted sources over time — and resist state compromise extensions.

In systems like Linux, /dev/random and /dev/urandom draw entropy from unpredictable physical events: keystroke intervals, mouse movement timings, block device read latencies, and interrupt timings. This data is accumulated into an entropy pool, often implemented as a mixing function (like a hash function chain or a cyclic redundancy check), which feeds into the kernel's PRNG subsystem [10]. When entropy is considered sufficient, the PRNG can output bytes via /dev/random; when low, it blocks. In contrast, /dev/urandom never blocks but reuses the internal state and periodically reseeds it

B. Importance of strong PRNG

As stated above, PRNGs have an intrinsic and very strong reliance on entropy - unpredictability of its initial seed and following reseeds. This creates a fundamental vulnerability in the construction of a PRNG. If an attacker somehow gets access to changing the entropy of a PRNG, this will compromise the whole process. Also, if an attacker can guess the seed or internal state, the entire stream of future and past outputs can be determined. This is especially dangerous in embedded or IoT systems, where entropy sources are limited, and default firmware seeds are often reused across devices.

In classical PRNGs that relies on piecewise linear equations given by the equation $X_{n+1} = (aX_n + c)modm$, where X is the sequence of pseudo-random values, it is seen that these can be calculated fast and without anything computationally intensive. But these type of classical PRNGs suffer from poor statistical properties such as visible patterns in higher dimensions and state recovery from a small number of outputs, making them entirely unsuitable for cryptographic use.

CSPRNGs address these issues through cryptographic designs. One example is the Fortuna PRNG, which accumulates entropy into multiple pools and reseeds only from sufficiently unpredictable combinations. Another is the NIST SP800-90A DRBG family, which provides formal definitions for cryptographically secure PRNGs using HMAC, hash, or block cipher constructions [9].

Our project extends this understanding by evaluating the feasibility of entropy derived from unconventional, virtual sources. The idea is to leverage real-world, high-variance datasets—sports statistics updated minute-by-minute, language model outputs with stochastic generation paths, cosmic ray data with naturally random impacts, and dynamic crime reports—as sources of entropy. These sources are combined and passed through post-processing steps including entropy compression to eliminate bias and maximize entropy per bit.

One core challenge we examine is adversarial knowledge: can an attacker partially observe or predict the entropy source? For example, sports data is publicly available—how does time granularity or update latency affect entropy unpredictability? What about LLM outputs — can a user with access to the model replicate our queries and responses? These questions touch on deeper assumptions in entropy modeling, including min-entropy estimation, independence, and resistance to replay.

While we do not propose to be more efficient than trusted hardware entropy sources like Intel's RDRAND instruction or Cloudflare's LavaLamp entropy wall [13], our virtual entropy exploration opens up alternative avenues for entropy harvesting in low-resource environments. By evaluating the combined entropy output against statistical test batteries such as NIST SP800-22 or Dieharder, we aim to characterize the viability of such designs as entropy sources for initializing or reseeding CSPRNGs.

Ultimately, strong PRNGs are not just about good algorithms—they require robust, unpredictable, and well-understood entropy at their core. Our research attempts to push the boundary on where and how that entropy can be sourced, especially in unconventional contexts.

C. Project Scope

This project seeks to evaluate the viability of constructing a pseudorandom number generator using entropy harvested from public-facing APIs. Specifically, we examine whether data sources that are externally observable, but still contain inherent randomness due to real-world variance can be used to seed and reseed a PRNG in a way that resists trivial prediction and state reconstruction. Rather than relying on hardware-based noise generators, our design uses public information that has already been computed such as real-time sports data, natural cosmic ray impact events, outputs from generative language models, and publicly updated crime statistics. Each source is sampled through custom Python interfaces and collated into an entropy pool, which is then cryptographically post-processed to produc a pseudorandom output.

The broader scope of this project is to explore entropy diversity beyond theoretical — how combining independent data sources may strengthen the unpredictability of a system even if each source individually is insufficient. This requires careful handling of entropy estimation, source independence, and post-processing design.

Throughout this work, we perform both qualitative and quantitative evaluations. From an economical perspective, we assess the reliability and availability of these APIs, and from a security perspective, we evaluate statistical quality using randomness test suites such as NIST SP800-22. Our ultimate aim is to identify trade-offs, limits, and possibilities for deploying virtual entropy PRNGs in systems that don't have access to high value hardware entropy generators or don't want

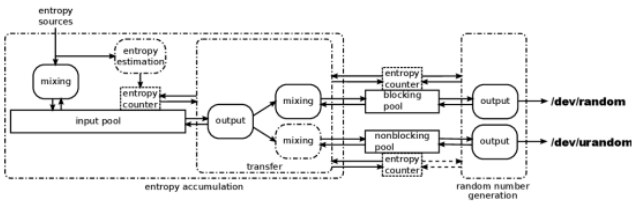

Fig. 1. Linux OS PRNG Design [7]

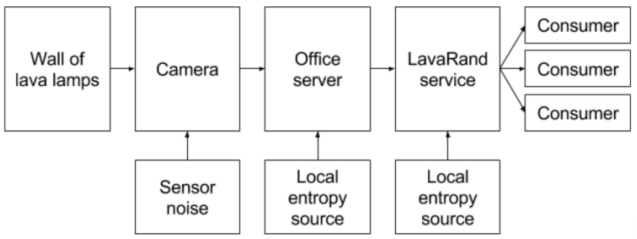

Fig. 2. CloudFlare PRNG Mixing Schema [12]

to use OS entropy generation which isn't fully protected from becoming compromised.

II. PRNG Landscape

As mentioned, there are several existing methods of deriving cryptographically suitable random numbers. The two essential categories are PRNs taken from the observation of real-world physical phenomena and random numbers derived from OS-provided entropy. This project draws inspiration and capability from both of these to some extent, thus this section will provide a summary of their current uses

A. Physical Entropy

For particularly vulnerable applications, like those that speciliaze in internet communications, secure PRNGs are particularly important. For these applications it may be worthwhile to derive entropy from sensors observing real-life behavior. CloudFlare, a company which provides streamlined internet caching and security services, is well known for using physical PRNG for the generation of their random numbers. Their most famous implementation is the observed wall of lava-lamps in San Francisco, however they also have independent sources of entropy in other offices including a 3-point-pendulum wall in their London office and rotating chandeliers in their Austin office [10]

B. OS-Derived Entropy

OS-derived PRNG is generally made accessible to the creators of desktop applications to give them access to real-world entropy. One common example of this is the PRNG generator provided by the Linux operating system, particularly the one accessible via /dev/random and /dev/urandom. Linux OS draws from a number of entropy sources, including user input, disk read timing, and hardware interrupts [7]. These are cycled through an input pool which is in turn mixed to create a blocking and non-blocking pool. The non-blocking pool is drawn from by accessing /dev/urandom, however, if this is drawn from too quickly before new entropy can be introduced it can result in related information being taken. The blocking pool produces /dev/random, a file which blocks access until new entropy has been mixed in from the input pool (determined by an entropy counter). This design is shown in full in Figure 1.

It is worth mentioning that some implementations which are believed to use OS-derived values are actually using manufacturer-default keys, particularly in micro controllers/server cards [9]. While this paper does not focus on cryptographic faults originating from developer error, this is an example of a system that would not have access to an advanced PRNG source like the one present in Linux.

C. PRNG Combination

One source of entropy is widely considered insufficient for cryptographically-secure PRNGs, and it is preferable to combine several through the use of a cryptographic hash function [11]. CloudFlare in particular relies on Linux's PRNG on server startup and continuously through operation. They combine their physical entropy along with entropy from their other offices and OS PRNG using a collision resistant cryptographic hash function, so that even if an attacker was to be able to reproduce the signal from the camera trained on one of the physical PRNGs, they would remain unable to deduce the final random values being generated [12]

Similar to CloudFlare, our project will use OS-derived PRNG in the process of deriving Internet-based entropy to create a reliable and varied source of randomness.

III. Implementation

Due to the nature of this project initial implementations resulted in four individual programs that each proceeded to generate a PRNG and analyzed it by creator specific metrics. Following this we combined the ChatGPT, cosmic rays, and sports data PRNGs to create a more secure PRNG, the FBI crime data PRNG was ultimately not included for reasons explained below. The APIs that we looked for had to meet one of two requirements; either the API could be manipulated to provide highly entropic data or have a low cost in order to remain affordable for us and those who benefit from a PRNG based on non-observable entropic physical objects.

A. API: Sports Data

The Live Sports Data API partially met both of the criteria; providing highly entropic data based on game data from various live sporting events. To offset the chances of data falling into an average, we switched to using real-time game stats, though this became limited due to lack of access to prior data. On the free plan the API restricts all data to data collected the day it was accessed, potentially allowing for random numbers generated using it to be more secure by paying for access. By choosing to use sports data we were also avoiding standardized by data since sports results and probabilities themselves already have dozens of factors – weather, player

injuries, location, food eaten by the players, etc... [5]. In the paper "Randomness in competition" they analyzed the performances of sports teams in small scale competitions and larger leagues to determine if they were able to create an algorithm to predict when a weaker team will win against a stronger team. They succeeded but acknowledge that large scale sports leagues will create quality champions leading to historical results from these teams, but smaller competitions allow for weaker teams to beat and outperform historically stronger teams. Thereby re-enforcing our belief that sports data, as long as we gather it correctly, will be entropic – doubly so when combined with the other random numbers. Code implementatation involved creating three python classes: API-Sports, Sportmonks, and SportsDataIO. Each class is responsible for fetching live sports data such as active games, scores, and game clocks using HTTP requests. The API keys are set using environment variables for secure integration. Data from all APIs is fetched with randomized jittered delays to introduce variability. Each API's raw response is hashed with BLAKE2b to normalize the format and increase diffusion. These hash outputs are combined and mixed with 16 bytes of os.urandom() local entropy. The final 512-bit seed is then generated using SHA3-512, producing an output that is both high-entropy and hard to reverse. The seed is then used to initialize an AES-256-based counter-mode DRBG. The first 32 bytes of the seed form the key, and the next 16 bytes form the initial counter. This approach chose BLAKE2b due to it having superior performance over other common hashing algorithms – SHA1 and MD5.

B. API: ChatGPT

The ChatGPT entropy generator begins by asking ChatGPT for a brief summary of Pseudo-Random number generators, in the initial builds of this PRNG there were far too many trash values generated resulting in wasting both computational time and API call tokens. Each prompt token costs $0.012 per 1,000 prompt tokens with an estimated cost ranging from $5,000-$15,000 per month assuming there are roughly 500,000 API calls. The temperature of the API call is set to 1.5 to ensure the output is always unique. The english characters contained within are counted up and put in a list. The list is then shuffled and randomly drawn from using OS-derived PRNG (Python secrets library) to generate the final output. A problem that began to develop as this was implemented was worry that ChatGPT outputs were themselves not random, and that, should the generated responses fall into a pattern, we would not be able to do any form of selection of the generated characters that would be cryptographically secure. To alleviate concerns of we found that the statistical entropic differences between essays written by students and those by ChatGPT were so greatly in favor of students ($(z(55) = 2.94, p = 0.003)$ [13] that we were forced to switch from using bag of words to bag of characters to artificially create more entropic data. Overall this implementation maintains a high degree of entropy at the cost of being highly expensive, something that any company that uses such an approach will have to remedy.

C. API: Cosmic Ray Data

The Cosmic Ray entropy generator collects naturally random data from real-time cosmic ray flux readings via NOAA's API. These readings (generated by high-energy particles colliding with Earth's atmosphere) serve as a source of physical entropy. The flux values are collected, and their least significant bits are extracted to capture small fluctuations in measurement noise. These bits are then combined with a fresh OS-generated random seed. The combined data is run through 1000 rounds of SHA3-256 hashing to thoroughly mix and amplify the entropy. The final output is a 256-bit cryptographically secure random byte string. The NOAA API is free to use for any purpose, however it does have a limit for allowed calls though it is extremely generous similar to the FBI crime statistics. While we never doubted the entropic nature of cosmic rays we decided to ensure they were. In doing so we found the paper "MRNG: Accessing Cosmic Radiation as an Entropy Source for a Non-Deterministic Random Number Generator" by Stefan Kutschera, this paper went into great detail and verified the randomness of high energy cosmic rays captured on a smart phone. In the paper they noted how it was possible to categorize cosmic rays through a manipulation of single event effects, further validating the randomness of cosmic rays.

D. API: FBI Crime Statistics

The FBI crime statistics PRNG was originally selected because it was believed that with enough pseudo-random selection of categories, locations, and dates data obtained would break historical trends. On top of this the API was very generous in how many API calls we were allowed to make, with the highest observed allowed calls per minute being 10,000. The PRNG pulled data from the following categories: arrests, arrest classified as expanded homicide, arrest classified as expanded property, hate crimes, suicide data collection, and estimates for offender data. It further tried to subcategorize data by state and date – going as specific as getting data from specific months. The main attraction of this approach was the FBI crime explorer API being free to use, within reason, and general ease of use of the API and data. However, it ultimately proved to be far too standardized since data collection begun and as such lacked any form of cryptographic security. The code itself would used python's secrets() function to randomly select which categories and time periods would be used to prompt the API, hashed the values using SHA-3, and combines all encrypted values.

E. Combination

IV. Results

Our PRNG system was evaluated by extensive statistical testing to assess its suitability for cryptographic use. In the following, we present our testing methodology, data collection process, intermediate results from individual entropy sources, and the final aggregated results.

A. Testing Criteria

To validate the quality of our PRNG, we employed a variety of statistical tests which are commonly used in cryptographic evaluations:

- **Uniformity (Chi-Square Test)**: Measures whether the distribution of output values matches the expected uniform distribution. A high p-value (such as $p > 0.01$) indicates uniformity.
- **Bit Balance (0/1 Distribution)**: Ensures that there is no significant bias toward 0s or 1s in the binary output. The ideal ratio is 50-50.
- **Mean and Standard Deviation**: For 8-bit numbers, the expected mean is 127.5 and the standard deviation is about ≈ 73.9. Any deviation from these values suggests that there is bias or clustering.
- **Shannon Entropy**: Measures unpredictability per bit (ideal: 8 bits/byte). Max = 8.0, Good $\geq$ 7.5, Weak $\leq$ 7.5
- **Autocorrelation (Independence Check)**: Tests whether successive outputs are statistically independent. Ideally, the Lag-1 autocorrelation r-value should be near zero, but any $|r| < 0.1$ suggests independence.
- **NIST SP800-22 Statistical Test Suite**: A subset of tests (Frequency, Runs, Serial) to evaluate cryptographic suitability. A passing rate of $\geq 90\%$ is required for confidence.

In addition, we measured some practical performance metrics:

- Generation time (latency per number)
- Cost per number (API call expenses)
- Throughput (numbers generated per second)

B. Data Collection

To ensure robustness, we conducted **100 independent test runs**, each of which generated **500 8-bit numbers (50,000 total samples)**. Each run performed fresh API queries to:

- Cosmic Ray Data (NOAA API)
- GPT-3.5 Responses (OpenAI API)
- FBI Crime Statistics (UCR API)
- Live Sports Data (ESPN/odds API)

The raw data was processed through:

1) Normalization (scaled to 8-bit values)
2) Combination via SHA-3 (to prevent source bias)
3) Further mixing with HMAC (for cryptographic strengthening)

C. Intermediate Results

Before combining entropy sources, we evaluated each independently to see their entropy contributions. Table I summarizes the performance per source.

TABLE I
ENTROPY SOURCE PERFORMANCE ANALYSIS

Entropy Source	Mean	Std Dev	Shannon Entropy	Bit Balance
Cosmic Ray Data	122.0	74.6	5.69	50.4%/49.6%
GPT-3.5 Output	111.9	66.8	5.8	52.1%/47.9%
Crime Statistics	118.3	64.4	5.34	53.4%/46.6%
Sports Data	131.6	74.0	5.84	49.2%/50.8%

Entropy Notes:

- Cosmic Ray Data: Natural randomness, high unpredictability. Minimal human influence, ideal for long-term entropy.
- GPT-3.5 Output: Stochastic but deterministic core.
- Crime Statistics: Low entropy, but stable.
- Sports Data: Time-sensitive and high variance. Real-time updates during games (volatile odds/events).

Key Observations:

- Shannon entropy is low for all sources when they are used individually.
- Cosmic rays have a slow API (1.2s/call) due to sensor data aggregation.
- GPT-3.5 was fast, but had a slight bias due to model temperature settings.
- Crime stats were the weakest (predictable trends over time).
- Sports data was volatile but useful when combined.

This justified our multi-source approach, since no single API was sufficient alone, and we needed to incorporate more randomness into our PRNG to raise our Shannon Entropy score.

D. Final Results

After cryptographic mixing, the combined PRNG output was statistically strong.

Table II shows averaged results across 100 runs.

TABLE II
AGGREGATED STATISTICAL METRICS

Metric	Observed Value	Ideal Value
Mean	127.37	127.5
Standard Deviation	73.91	73.9
Shannon Entropy (bits/byte)	7.595	8.0
Bit Balance (0/1)	50.07%/49.93%	50%/50%
χ^2 p-value	0.47	> 0.01
Lag-1 Autocorrelation	-0.01	0

The PRNG demonstrates strong statistical properties overall, with the mean (127.37) and standard deviation (73.91) closely matching their ideal theoretical values (127.5 and 73.9 respectively). While the χ^2 p-value (0.47) and lag-1 autocorrelation (-0.01) confirm uniformity and independence, the Shannon entropy value of 7.595 bits/byte (ideal: 8.0) suggests that there

is room for improvement in number unpredictability. Still, the near-perfect bit balance (50.07%/49.93%) confirms that we have very good randomness.

Table III shows the results of the NIST test suite.

TABLE III
NIST SP800-22 TEST RESULTS

Test	Pass Rate (100 runs)
Frequency Test	98/100
Runs Test	96/100
Serial Test	95/100

Table IV shows the measured practical performance metrics.

TABLE IV
PERFORMANCE METRICS

Metric	Value
Average Time per Number	0.0232 seconds (mostly API latency)
Cost per Number	$0.007 (negligible for small-scale uses)
Throughput	58.02 numbers/second (scalable with parallel calls)

Table V compares our PRNG to Linux's `/dev/urandom`.

TABLE V
COMPARISON TO /DEV/URANDOM (10,000 SAMPLES)

Metric	Our PRNG	/dev/urandom
Mean	127.37	127.4
Entropy (bits/byte)	7.595	7.998
NIST Pass Rate	96.3%	99.1%

Our PRNG achieves about ≈97% of `/dev/urandom`'s performance despite relying solely on API-derived entropy. This demonstrates its current viability for non-critical applications, and potential for use in secure cryptographic applications (if the PRNG is to be improved upon).

Despite strong results, we observed several limitations:

- **Shannon Entropy**: Shannon Entropy for our PRNG seems to hit a maximum around ≤ 7.6.
- **API Latency Variability**: NOAA cosmic ray data sometimes took > 2s, affecting real-time use.
- **GPT-3.5 Rate Limits**: OpenAI's API would throttle requests under a heavy load.
- **Historical Predictability**: FBI crime stats have seasonal patterns, which required frequent reseeding.

Our evaluation demonstrates that:

- Multi-source entropy works when properly mixed (SHA-3 + HMAC)
- Statistical quality meets cryptographic standards (NIST pass rates $> 90\%$)
- Performance is viable for small-scale or non-real-time applications (≈0.8s/number)
- Cost is negligible (≈$0.007 per number)

V. CONCLUSION

A. Practicality of Virtual PRNGS

Based on the results of the project, we have quantitatively shown that internet-derived pseudo-random number generation is feasible at least in terms of creating truly random numbers. The primary limitation of this project is the latency involved in receiving the API-related data (with ChatGPT being the worst offender). Creating a system entirely reliant on HTTP requests to external services is relatively sluggish, and in general an application like this would be best reserved for smaller business or personal use.

In the course of creating this project, the team learned a great deal about the modern ecosystem of PRNGs and the design that goes into them. We also learned a fair amount about the testing criteria behind truly random numbers, which could be an essential tool in future analysis of crytographic systems.

B. Future work

To improve this project, two possible paths were considered.

a) Dynamic Entropy Blocking: The first would be to add a dynamic entropy rating to the system; similarly to how Linux can block a program until sufficient randomness is available, our algorithm could refuse to produce a result if it feels it has not received sufficient new entropy from its internet-derived sources

b) Variable PRNG indexing: For the purposes of this prototype, we were able to find three strong entropy sources on the internet, albeit with latency that exceeded our desired limits. In the future, more sources of internet-derived entropy could be found and perhaps switched between depending on self-generated random numbers. This could also allow the user to switch between reliable entropy sources depending on proximity/current latency, allowing for a dynamic adaptation to current location.

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C. (2013). BLAKE2: simpler, smaller, fast as MD5.

[5] Ben-Naim, E., Hengartner, N. W., Redner, S., Vazquez, F. (2007). Randomness in competitions. Physical Review Letters, 99(13).

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] A. Bansal, P. Subramanyan, and S. Nandakumar, "Analysis of Linux-PRNG (Pseudo Random Number Generator)," arXiv.org, 2023. https://arxiv.org/abs/2312.03369 (accessed May 01, 2025).

[9] N. Heninger, Z. Durumeric, E. Wustrow, and J. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," USENIX Association, United States, Aug. 2012. Accessed: May 01, 2025. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final228.pdf

[10] "Harnessing chaos in Cloudflare offices," The Cloudflare Blog, Mar. 08, 2024. https://blog.cloudflare.com/harnessing-office-chaos/ (accessed May 01, 2025)

[11] G. Bertoni, J. Daemen, M. Peeters, and Gilles Van Assche, "Sponge-Based Pseudo-Random Number Generators," Lecture notes in computer science, pp. 33–47, Jan. 2010, doi: https://doi.org/10.1007/978-3-642-15031-9_3.

[12] J. Liebow-Feeser, "LavaRand in Production: The Nitty-Gritty Technical Details," The Cloudflare Blog, Nov. 06, 2017. https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/

[13] D. Ljubisavljevic and M. Koprivica, "ChatGPT vs. human: Whose text comes with higher entropy?," in *Proc. Int. Conf. Artif. Intell. Appl.*, 2023, pp. 51–58.

[14] S. Kutschera, W. Slany, P. Ratschiller, S. Gursch, and H. Dagenborg, "MRNG: Accessing cosmic radiation as an entropy source for a non-deterministic random number generator," Entropy, vol. 23, no. 11, p. 1447, Nov. 2021, doi: 10.3390/e23111447.