*International Journal of Basic and Applied Sciences*, 14 (8) (2025) 382-398

International Journal of Basic and Applied Sciences

*Website: www.sciencepubco.com/index.php/IJBAS*
*https://doi.org/10.14419/9bppfw31*
***Research paper***

# Towards Generation of Secure Pseudorandom Prime Dataset Using Generative Adversarial Networks and The Learning Parity with Noise Problem

**Daniel Asiedu [1] *, Patrick Kwabena Mensah [1], Peter Appiahene [2], Peter Nimbe [1]**

*[1] Department of Computer Science and Informatics, University of Energy and Natural Resources, Sunyani, Ghana*
*[2] Department of Information Technology and Decision Sciences, University of Energy and Natural Resources, Sunyani, Ghana*
**Corresponding author E-mail: daniel.asiedu.stu@uenr.edu.gh*

*Received: November 10, 2025, Accepted: December 10, 2025, Published: December 17, 2025*

**Abstract**

At present, prime numbers are an essential part of online security for secure key generation and digital signatures due to their unpredictability and resistance to factorization attacks, which has led to considerable interest in research on distributions of primes, especially prime gaps and hidden patterns. Traditional prime analytical methods lack structural insight at the scale of cryptographic systems. This has led to greater interest in applying machine learning (ML) techniques to prime analysis. However, due to a lack of publicly available high-quality datasets, the evaluation of prime behavior using ML is often impeded. Furthermore, current prime generation techniques tend to become computationally inefficient when scaled to produce large quantities of high-bit-length prime numbers. In response, this paper proposes a novel generative pipeline, GANLPN, that leverages Generative Adversarial Networks (GANs) combined with the Learning Parity with Noise (LPN) problem to generate bulk primes that are cryptographically secure. In this way, a publicly accessible dataset of 1,115,000 1024-bit primes has been generated. The generated sequences passed all NIST SP800-22 statistical randomness tests with better inference time and throughput. The present work outlines a path for future ML-assisted studies of prime patterns, gaps, distributions, and the presence of weak keys in cryptography.

***Keywords***: *Cryptography; Generative Adversarial Networks; Learning Parity with Noise; Prime Dataset; Pseudorandom Number Generator.*

## 1. Introduction

Prime numbers are well-known and important in number theory, and are especially significant in contemporary cryptography. A key use is in public key systems, whose security relies on the infeasibility of factoring the product of two large prime numbers. Primes also materialize in hash functions, digital signatures, and random number generators for maintaining secure digital communication. How primes are unpredictably distributed makes them great for encryption methods that need strong security. Primes are also helpful in coding theory, error detection, and complex mathematical simulations. Their unique complexity and non-linear distribution of primes make them practically valuable and a subject of enduring interest in mathematics and computer science [1 - 5]. Prime numbers still make up a significantly sizeable area of theoretical mathematics research beyond application. Unresolved issues, such as the Goldbach Conjecture, the Twin Prime Conjecture, and the Riemann Hypothesis, concern the knowledge of the properties of prime numbers. Studying the distribution and pattern of prime numbers, particularly the behavior of prime gaps, remains one of the most difficult and unsolved challenges in number theory [6 - 10]. Prime numbers are not uniformly distributed, and their generation lacks predictable patterns. The unpredictable nature of their distribution makes it exceptionally difficult to model or predict prime occurrences. While prime unpredictability and randomness are excellent for keeping data safe, they also pose a challenge to researchers who want to study how primes are distributed. After years of mathematical argument, there is no model or deterministic formula to produce the precise location or distance between two consecutive primes. Therefore, understanding prime densities continues to expose profound mathematical knowledge and incite progress in algebra, analysis, and topology. Although theoretical discoveries, such as the Prime Number Theorem, describe asymptotic distributions, actual analyses of the local patterns and gaps of primes must be based on the analysis of actual prime sequences over various scales. Regardless of whether proofs or statistical sampling of relatively small primes are used, traditional avenues will not generate the complexity of the behaviors of large primes, which is relevant to cryptography. One way to approach the complexity of these problems is through the use of artificial intelligence (AI). These days, the type of machine learning that can find patterns in complex mathematical problems is deep learning (DL). DL models are particularly adept at recognizing complex patterns and relationships in large datasets, which presents the opportunity to examine the hidden structure of prime sequences more closely.

A key hurdle faced in using AI and ML to study primes is the lack of diverse, high-quality, large public datasets for prime numbers. Most datasets are deterministic, and the variety of primes is so limited that they are not helpful for training models that will discover latent structures or patterns in primes. Classical prime generation methods generate primes within a specific range but are not scalable and typically lack variety for AI pattern recognition or robust security in cryptography. These algorithms are designed for generating and verifying individual primes, rather than batch prime generation with either structural or statistical consistency [11], [12]. Therefore, there is a critical need for processes that produce pseudorandom prime datasets, which have the unpredictability of true primes but are generated in a statistically strong and secure manner.

Our work addresses this gap by introducing the first large-scale, publicly available dataset of 1024-bit primes that could be used for ML analysis. This study presents a novel pipeline for generating a secure and structured dataset of large prime numbers using a Generative Adversarial Network (GAN) with the Learning Parity with Noise (LPN) problem, hereinafter referred to as GANLPN. Generative Adversarial Networks, or GANs, are popular among the emergence of neural networks because they can create data that closely resembles a specific type of data. A GAN comprises the generator and the discriminator, trained in a minimax game. The generator learns to produce realistic samples. In contrast, the discriminator learns to distinguish between real and generated data. This adversarial process encourages the generator to create highly indistinguishable and diverse outputs. Since we are dealing with cryptographic systems, specifically pseudorandom number generation (PRNG), GANs offer a good approach to learning and modeling complex bit distributions that are hard to reverse-engineer or predict.

Many researchers have examined the LPN problem, one of the complex computational problems in cryptography. It involves learning a secret binary vector given random linear combinations of its bits with added noise. The LPN assumption holds up against classical and quantum attacks, making it a strong foundation for cryptographic applications. PRNGs using LPNs create outputs that are theoretically based on complex problems, thus providing proven security guarantees [13], [14]. By combining GANs with the LPN problem, we utilize the advantages of these paradigms: the statistical learning capabilities of neural networks, in conjunction with the intractability assumptions of cryptographic theory. In this pipeline, the GAN learns to approximate the bit distributions generated by the LPN process. The pseudorandom outputs get their statistical nature from GAN training and their unpredictability from LPN. This mix improves the quality of random numbers, making it more challenging to spot patterns or predict outcomes. This method is excellent for cryptographic tasks, such as generating primes. Using GANs, the model can adapt and scale well, which is key for handling long pseudorandom sequences.

The main contributions of this study are:

A GAN model trained to produces high-entropy pseudorandom binary samples.

1) Integration of the LPN problem to embed cryptographic hardness into the generation process.
2) A deterministic post-processing algorithm that balances and normalizes pseudorandom binary outputs from the GANPLN to correct distributional skews.
3) An optimized incremental primality testing algorithm based on the Miller-Rabin test to map pseudorandom numbers to the nearest strong primes.
4) A reproducible pipeline for generating large datasets of primes.
5) The first publicly accessible 1,115,000 dataset of 1024-bit prime numbers that can support research in machine learning, number theory, and cryptography.

The pseudorandom prime dataset provided by this research opens new paths for a variety of advanced applications. Specifically, it enables training ML models to study the characterization of prime gaps over scales appropriate for modern cryptographic systems, thereby allowing researchers to study the statistical behavior of large prime gap intervals. Additionally, develop AI detectors that spot weak primes that may be more vulnerable to factorization attacks. This, in turn, helps with assessing cryptographic risks. By using data-driven methods on this large dataset, researchers can uncover new patterns in the distribution of primes that classical analysis methods have missed. This research is essential because of the increasing threats posed by advanced classical algorithms and the emerging challenges of quantum computing. Understanding how prime numbers behave, with the aid of AI, is crucial to developing the next generation of secure cryptographic protocols.

## 2. Literature Review

### 2.1. Prime numbers and their properties

A prime number is a natural number greater than one that is exactly divisible only by itself and the number 1. Although this definition may seem straightforward, the distribution of prime numbers is a complex situation that has intrigued many researchers in number theory. One interesting thing about primes is that they are not evenly spaced along the number line. Even though there are infinitely many prime numbers, they become less frequent as the numbers become larger. The Prime Number Theorem explains this by indicating how many prime numbers are below a certain number $x$ is approximately $x/\ln(x)$, where $\ln(x)$ is the natural logarithm of $x$. This asymptotic outcome provides insight into how the occurrence of primes diminishes as integers increase, resulting in a process that becomes progressively more challenging and costly in computation. Prime numbers have some essential features that matter in theory and practical applications. They allow for unique factorization where every positive whole number above one can be broken down into a unique set of primes (the Fundamental Theorem of Arithmetic). They also play a central role in modular arithmetic, Euler's totient function, and the structure of many algebraic systems, making them indispensable in cryptographic computations [15], [16].

### 2.2. Existing prime generation algorithms and their limitations

The process of generating prime numbers has long relied on various mathematical algorithms to find primes within a given range at a rapid pace. These algorithms have many applications in number theory and computational mathematics. However, these traditional approaches to generating primes, particularly for contemporary applications for cryptography that require secure, high-entropy, and large primes, have limits in scaling, unpredictability, and randomness properties. Some of the notable algorithms include:

Sieve of Eratosthenes: This method is among the oldest and most established algorithms for determining all primes $x$ in the range $2 \leq x \leq n$. It works mathematically by marking the multiples of each prime number, starting with 2 and proceeding upward. As an example, to find all primes less than or equal to 30, the sieve algorithm would start with 2 and mark out 4, 6, 8, …, 30. Then the algorithm would proceed to 3 and mark out 6, 9, 12, …, 30, and so on, until it found all the primes less than or equal to 30. The final list of unmarked numbers consists of the primes [17]. While the sieve is quite efficient for small values of $n$ (with a time complexity of $O(n \log(\log n))$), it requires

a large amount of memory when n is large. For example, counting primes of hundreds or thousands of bits is impractical in a modern cryptographic system.

Sieve of Atkin: A more modern and mathematically intensive variation of the Sieve of Eratosthenes. It relies on quadratic forms and modulo operations to filter out non-prime numbers. Specifically, it examines equations such as $4x^2 + y^2, 3x^2 + y^2$, and $3x^2 - y^2$, and uses modular conditions to determine whether a number should be flipped between prime and non-prime states. For example, numbers of the form $4x^2 + y^2$ where n mod $12 = 1$ or 5 are potential primes [18]. This technique improves performance for large numbers, particularly when compared to the Eratosthenes sieve, but it is more complex to implement. Moreover, like its predecessor, it is not inherently suited for generating secure primes, as it deterministically outputs all primes in a range without incorporating entropy or randomness.

Segmented Sieve of Eratosthenes: The segmented sieve is a space-efficient extension of the original Eratosthenes algorithm. Instead of sieving all numbers up to n at once, it processes the range in fixed-sized segments. This allows prime generation over larger ranges without requiring memory proportional to n. For instance, generating primes between $10^9$ and $10^9 + 10^6$ can be done using a precomputed list of base primes up to $\sqrt{10^9 + 10^6}$ and sieving just that segment [19]. While this segmentation makes the algorithm more scalable, it remains a deterministic method that generates primes in order. It cannot generate large, cryptographically strong primes with sufficient entropy or unpredictability, which are crucial for cryptographic security.

Sieve of Sundaram: A variant of the sieve of Eratosthenes, a simple deterministic algorithm to find all the prime numbers up to a given number. The sieve starts with a list of the integers from 1 to n. All numbers of the form $i + j + 2ij$ are removed from this list, where i and j are positive integers such that $1 \leq i \leq j$ and $i + j + 2ij \leq n$. The remaining numbers are doubled and incremented by one, giving a list of the odd prime numbers (that is, all primes except 2) below $2n + 2$ [20]. The Sieve of Sundaram is more memory-efficient than Eratosthenes when generating only odd primes and avoids even number processing altogether. However, it is typically slower and less practical for generating huge primes.

Wheel Factorization: Wheel factorization makes the sieving process easier by skipping over composite numbers. It sets up a repeating pattern, or "wheel", using the least common multiple (LCM) of small prime numbers like 2, 3, and 5. For example, by using this wheel, you can quickly avoid numbers that can be divided by these primes, which means you don't have to recheck them. Mathematically, the wheel of size 30 (LCM of 2, 3, 5) can help pre-filter out about 75% of numbers before applying primality tests [21]. However, wheel factorization is mostly a preprocessing optimization; it does not generate primes independently and lacks the randomness needed for secure prime generation.

## 2.3. Pseudorandom number generation based on GANs

Pseudorandom numbers (PRNs) are a series of numbers that seem to be random, but deterministic algorithms generate them. True random numbers are derived from fundamentally unpredictable physical processes, while pseudorandom numbers are derived from a seed or initial value by means of a mathematical formula or algorithm. Despite being deterministic, well-designed pseudorandom number generators (PRNGs) produce outputs that closely mimic the statistical properties of truly random sequences [22], [23]. The PRNs have significant applications across numerous areas, such as simulation, numerical analysis, gaming, machine learning, and, most importantly, cryptography. In cryptography, PRNGs need to be unpredictable. If a PRNG has patterns or is predictable, it can undermine the security of systems, for example, key generation, secure communications, and encryption. PRNGs are built on mathematical ideas like linear congruential generators, shift registers, and feedback systems.

Recent studies have investigated the use of ML models, particularly GANs, to learn complex distributions and generate pseudorandom sequences. Bernardi et al. were among the earliest researchers to look at the task of generating pseudorandom numbers using GANs that produce sequences statistically indistinguishable from random ones. Their study proposed two important working modes for GANs in PRN generation: the discriminative and predictive modes, which are now recognized as important contributions to modern PRNG design [24]. A subsequent study in the same area resulted in the development of a PRNG model based on the discriminative mode [25]. At the same time, Kim et al. incorporated recurrent neural networks (RNNs) to enhance statistical performance, achieving improved results on the NIST SP800-22 randomness test suite and adapting well to edge computing devices such as the Edge TPU [26]. Okada et al. further advanced the field by using the Wasserstein GAN with Gradient Penalty (WGAN-GP), significantly improving the quality of generated pseudorandom sequences [27]. Recent work has focused on fine-tuning GAN architectures by optimizing input-output parameters and network configurations, ensuring compliance with NIST SP800-22 standards under default settings [28]. Another evolution was a non-gradient learning algorithm based on a genetic algorithm that enhanced a GAN to optimize recursive PRNG performance and reliability [29]. The workflow improvement increases processing efficiency and establishes GAN techniques as a promising research avenue.

## 2.4. Classical and post-quantum based PRNGs

The RSA and discrete-log-based PRNGs assume that integer factorization or discrete logarithm computation in cyclic groups is complex. Suppose a quantum computer (via Shor's algorithm) breaks the one-way function that ensures the security of these PRNGs. In that case, an adversary using quantum technology could distinguish between outputs from a PRNG and outputs from true randomness. In effect, an adversary can recover an internal state, even with a finite number of tries. Therefore, classical number-theoretic generators cannot provide long-term security.

Post-quantum PRNGs build on mathematical problems believed to resist both classical and quantum algorithms. Their security does not rest on the hidden structure of a single, vulnerable algebraic group, but on issues such as finding short vectors in high-dimensional lattices (LWE), inverting random-looking systems of multivariate equations, or decoding random linear codes. Post-quantum replacements pay a heavy performance penalty for their quantum resistance: lattice-based generators are 10- 100x slower due to complex polynomial arithmetic, while even the fastest hash-based generators (SHAKE) are significantly slower than hardware-accelerated AES [30], [31], [32], [33], [34]. Table 1 summarizes key limitations as observed in both classical PRNGs and post-quantum PRNGs.

**Table 1:** Summary of Key Limitations of the Classical and Post-Quantum PRNGs

| Classical PRNGs | Post-Quantum PRNGs |
|---|---|
| • Blum-Blum-Shub (BBS) generators base their output on fixed recurrence relations, which leads to a rigid, non-adaptive design. The security of BBS generators can be completely undermined by Shor's algorithm, which efficiently factors **n**. They are impractical for any application that requires throughput greater than 100 bps.<br>• Micali-Schnorr generators also inherit RSA's vulnerabilities against Shor's algorithm due to the presence of a vulnerable public exponent, leaving the generator subject to attack.<br>• The Shor's algorithm efficiently computes discrete logarithms in cyclic groups, which breaks the Blum-Micali generator. The Blum-Micali generator yields only one unpredictable bit per modular exponentiation.<br>• A Fortuna PRNG based on Classical AES yields lower security against Grover's algorithm than AES-128 would, since Grover's algorithm reduces the total number of operations required to decrypt a message by a factor of 264.<br>• Linear Congruential Generators are fast, but they leak correlations in large-sample runs. The sequence eventually repeats (forms a cycle), and its structure can be analyzed, making it insecure for cryptography. | In Lattice-Based PRNGs:<br>• Security is susceptible to parameter choices (dimension n, modulus q, error distribution χ). Poorly chosen parameters can lead to devastating attacks. The correct parameters are still an active research area.<br>• Operations on secret vectors with small coefficients are highly susceptible to timing, cache, and power analysis attacks. Mitigations add overhead.<br>• Many designs require sampling error terms from discrete Gaussian distributions, which is computationally expensive and notoriously hard to implement securely against side-channels.<br>• In Hash-Based PRNGs:<br>• While hash functions are quantum-resistant, Grover's algorithm provides a quadratic speedup for preimage attacks. To achieve 128-bit classical security, you need a 256-bit hash output. Doubling key sizes and output lengths increases computational and bandwidth costs.<br>• Creating efficient stateless, deterministic randomness requires complex tree structures (as in SPHINCS+), with significant storage or computational overhead.<br>• In Multivariate Quadratic (MQ) Based PRNGs:<br>• Many multivariate cryptosystems have been proposed and subsequently broken by sophisticated algebraic attacks (such as XL or Gröbner basis methods), requiring careful parameter selection and design to ensure robustness.<br>• A significant disadvantage is often the large public and private key sizes required to achieve sufficient security levels. |

### 2.5. Research gaps

- Classical and post-quantum PRNG outputs are deterministic structures. Neither family incorporates an adaptive mechanism to detect and correct statistical irregularities or bias in outputs, a requirement when producing large, bulk random PRNs.
- Post-quantum generators ensure the cryptographic security of message content by consuming substantial computational resources, and they sometimes exhibit repeating patterns (i.e., bias) detectable in finite-sample statistical tests.
- Classical PRNGs can produce high-quality randomness, but classical generators have no quantum resistance.
- Both classical and post-quantum approaches lack feedback loops or adversarial training that can self-adjust randomness quality across long sequences.
- Though classical algorithms for generating primes have been necessary, they suffer from scalability problems and do not yield unpredictability and randomness, both of which are essential for a secure prime dataset at scale.

In this study, the GANLPN method overcomes these limitations. First, GANs allow the model to learn and produce pseudorandom binary sequences of high entropy. Next, introducing the LPN problem to the generation process increases the output security of the derived streams. A deterministic bit-balancing algorithm corrects any imbalance that could be inherent in the GANPLN-generated binaries, thus providing uniformity and balance to the pseudorandom bitstreams. The post-processed binary outputs are then converted into integers. Finally, an enhanced Miller-Rabin test and an incremental search method take the final integers and determine the nearest prime numbers. This multistep method has proved to be a strong and secure way to generate large-bit-length prime numbers that exceed the limitations of traditional methods.

## 3. Methodology

We start by clarifying the central concept of the prime dataset pipeline established on GANLPN: the structural design of both the generator and the discriminator (LPN in our case) networks, and a detailed description of our proposed GANLPN-based PRNG training process. Followed by a deterministic bit-balancing algorithm for uniformity and the removal of distributional skew in the generated binary sequences, a binary-to-integer conversion, and an optimized Miller-Rabin primality test with an incremental search for the closest cryptographically significant primes.

### 3.1. Proposed GANLPN PRNG framework

The A possible method for constructing a PRNG is to employ one-way functions (OWFs) along with their natural hardcore predicates [35], [36]. After being seeded with s0, the PRNG applies the OWF to the seed, and in each iteration, it applies the hardcore predicate to generate the output bitstream. Our framework uses the GANLPN to emulate both the one-way function and the hardcore predicate, effectively performing the PRNG's basic operations. Formally, a traditional PRNG is characterized by the tuple (S, μ, U, f, g), where:

- S: The state space.
- μ: The initial state distribution.
- U: The output space.
- f: The state transition function.
- g: The output function.

We map these components to the GANLPN-based PRNG implementation in our model as follows:

- The state space S represents all possible states of the PRNG. In our case, the state is defined by the secret key s and the noise vector e in the LPN problem. The state also includes the internal parameters of the generator (the neural network weights and biases).

- The initial state distribution μ defines how the initial state is sampled. In our case, the secret key s is sampled uniformly randomly from the space of binary vector size n. The noise vector e is sampled, so each bit is 1 with probability η (noise rate) and zero otherwise. The generator's parameters θ are initialized randomly (using PyTorch's default initialization).
- The output space U represents all possible outputs of the PRNG. In our case, the output is a 1024-bit binary vector (interpreted as a pseudo-random number).
- The state transition function f defines how the state evolves. In our case, the state transition involves updating the generator's parameters θ during training. The secret key s and noise vector e remain fixed during generation but are updated during re-initialization.
- The output function g defines how the state is mapped to the output. In our case, the output is generated by the generator network G, which takes a noise vector z as input and produces a 1024-bit output G(z). The output is threshold at 0.5 to produce a binary vector.

**Table 2:** Summary of PRNG Structure Mapping

| Traditional PRNG component | GANLPN-based PRNG implementation |
|---|---|
| State Space (S) | $(s, e, \theta)$: Secret key, noise vector, and generator parameters |
| Initial State Distribution (μ) | $s \sim \text{Uniform}(\{0,1\}^n)$, $e \sim \text{Bernoulli}(\eta)^m$, $\theta \sim$ Initialization |
| Output Space (U) | $\{0,1\}^{1024}$: 1024-bit binary vectors |
| State Transition Function (f) | $\theta \leftarrow \theta - \eta\nabla_\theta \mathcal{L}(G)$: Update generator parameters during training |
| Output Function (g) | G(z): Generator network producing 1024-bit outputs |

### 3.1.1. Generator design

The generator network is a neural network intended to generate PRN; that is, it takes a low-dimensional input noise vector and produces a high-dimensional output, producing bit sequences that mimic the statistical properties of true randomness. This architecture inspires deep feedforward neural networks and employs multiple fully connected layers with nonlinear activations to progressively refine the output distribution.

As shown in Fig. 1, the generator is a feedforward neural network comprising several fully connected (dense) layers with nonlinear activation functions designed to produce 1024-bit pseudo-random numbers from a 256-dimensional noise vector. The generator takes a 256-dimensional noise vector z as input. This noise vector is sampled from a Gaussian distribution ($z \sim \mathcal{N}(0,1)$), ensuring the input is random and diverse. The input layer consists of 256 neurons, depending on the dimensionality of the noise vector. The architecture has a hierarchical structure where each layer increases the feature representation of the input noise vector:

- Input Layer: A 256-bit random noise vector is fed into the network.
- Hidden Layers: The network comprises three fully connected layers with increasing dimensions (512, 1024, and 2048 neurons, respectively), each followed by a ReLU activation function defined as $\text{ReLU}(x) = \max(0, x)$, introduces non-linearity and helps the network learn complex patterns.
- Output Layer: The final layer maps the transformed representation to a 1024-bit output using a Sigmoid activation function defined as $\sigma(x) = \frac{1}{1+e^{-x}}$ to constrain the values between 0 and 1.

In a dense layer, every neuron is linked to all the neurons from the previous layer. The connections have weights, and each neuron includes a bias term. During training, the weights and biases are updated for the optimal network performance. The mapping through the generator can mathematically be described as follows:

Given an input noise vector $z_i \in \mathbb{R}^{256}$, the generator applies the following sequence of transformations:

$$h_1 = \text{ReLU}(W_1 z_i + b_1), W_1 \in \mathbb{R}^{512\times256}, b_1 \in \mathbb{R}^{512}$$

$$h_2 = \text{ReLU}(W_2 h_1 + b_2), W_2 \in \mathbb{R}^{1024\times512}, b_2 \in \mathbb{R}^{1024}$$

$$h_3 = \text{ReLU}(W_3 h_2 + b_3), W_3 \in \mathbb{R}^{2048\times1024}, b_3 \in \mathbb{R}^{2048}$$

$$\mu_i = \text{Sigmoid}(W_4 h_3 + b_4), W_4 \in \mathbb{R}^{1024\times2048}, b_4 \in \mathbb{R}^{1024}$$

Where:
$W_i$ are the weight matrices,
$b_i$ are the bias terms,
$h_i$ represents the intermediate activations, and
$\mu_i$ is the final output.
The output ($\mu_i$) is a 1024-dimensional vector, where each element represents a probability value between 0 and 1 using the sigmoid activation function. The sigmoid output has a threshold of 0.5 to produce a binary vector corresponding to the 1024-bit as follows:

$$output_i = \begin{cases} 1, & \mu_i > 0.5 \\ 0, & otherwise \end{cases}$$

This architecture is particularly suited for PRNG applications, where high-dimensional bit sequences must be generated with controlled randomness.

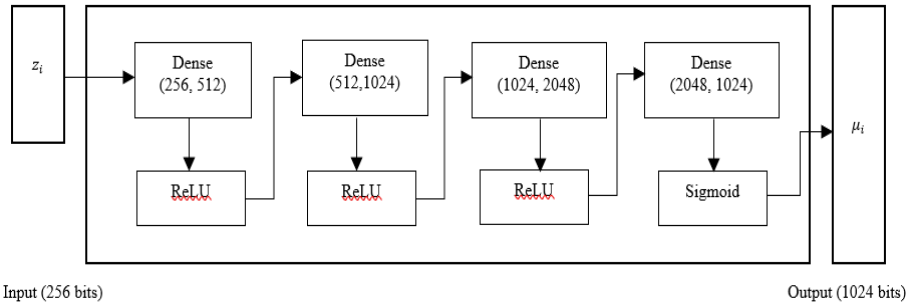


**Fig. 1:** Architecture of the Generator.

### 3.1.2. LPN problem

The LPN problem is a computational problem that is believed to be hard to solve. Combined with deep neural networks, it enhances randomness quality and resistance to attacks, making it ideal for PRNG design and other cryptographic applications.
Mathematical definition
Given:

- A binary matrix $A \in \{0,1\}^{m \times n}$,
- A binary vector $s \in \{0,1\}^{n}$,
- A binary noise vector $e \in \{0,1\}^{m}$, where each bit of e is 1 with probability $\eta$ (noise rate) and 0 otherwise,
- A binary vector $b \in \{0,1\}^{m}$ computed as: $b = A.s + e \pmod 2$

The goal is to recover the secret s given A and b. The LPN problem is complex because the noise term e makes it challenging to solve for the secret vector s using traditional linear algebra techniques. Even when matrix A is known, the added noise e introduces randomness and uncertainty, making it computationally hard to recover s accurately. This randomness complicates the problem and helps the LPN work well for cryptographic uses. The noise e keeps the outputs unpredictable and challenging to break into, which is essential for security. The difficulty of the LPN problem means it's hard to guess what the generator will produce. The noise e in the LPN problem introduces true randomness into the training process. The generator learns to produce outputs that mimic this randomness, ensuring high-quality pseudo-random numbers. After completing the training phase, the generator G can function as a PRNG.

## 3.2. Proposed GANLPN PRNG training design and prime generation

Algorithm 1 outlines the training procedure in three stages: initialization, generator training, and PRN generation.
The training process involves two main components:

- LPN Problem: Generates samples b that the generator tries to mimic.
- Generator: A neural network that learns to produce outputs G(z) similar to the LPN samples b.

The LPN problem provides the target outputs for the generator. Specifically, the LPN samples b is used as the ground truth during training. The generator is trained to reduce the gap between what it produces, G(z), and the LPN samples, b. The goal is to train the generator such that its outputs are indistinguishable from the LPN samples, ensuring cryptographic security and randomness.

### 3.2.1. Generator training

The generator is trained to produce outputs G(z) that are similar to the LPN samples b as follows:
Step 1: Initialize the generator
The generator is a feedforward neural network with a 256-dimensional noise vector z as input, fully connected layers with ReLU activation (except the output layer, which uses sigmoid), and a 1024-dimensional vector G(z) as an output (see Fig. 1).
Step 2: Generate LPN samples
For each batch of training:

- Generate a random matrix A.
- Compute the LPN samples $b = A.s + e \pmod 2$.

Where:
A is a binary matrix of size $m \times n$, randomly sampled from $\{0,1\}^{m \times n}$
s is a secret binary vector of size n, randomly sampled from $\{0,1\}^{n}$,
e is a noise vector of size m, where each bit is 1 with probability $\eta$ (noise rate) and 0 otherwise.
Step 3: Forward pass
Pass the noise vector z through the generator to produce the output G(z).
Step 4: Compute loss
The loss function is Binary Cross-Entropy (BCE), which measures the difference between the generator's output G(z) and the LPN samples b:

$$\mathcal{L}(G) = -\frac{1}{m}\sum_{i=1}^{m}[b_i \log(G(z)_i) + (1 - b_i)\log(1 - G(z)_i)]$$

Where:
$b_i$: the i-th bit of the LPN sample b.
$G(z)_i$: the i-th bit of the generator's output
Step 5: Backpropagation
Compute the gradients of the loss with respect to the generator's parameters $\theta$: $\nabla_\theta \mathcal{L}(G)$
Update the generator's parameters using gradient descent: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(G)$, where $\eta$ is the learning rate.
Step 6: Repeat

Repeat the process for multiple epochs until the generator's outputs $G(z)$ are indistinguishable from the LPN samples b.

Algorithm 1: Proposed GANLPN PRNG training design

Require:
Random seed s (secret key),
Generator $G_\theta$,
LPN parameters: matrix A, noise rate $\eta$,
Epochs EPOCHS,
Batch size BATCH_SIZE
1: Initialization:
2: Initialize weights and biases of $G_\theta$
3: Initialize secret key s (random binary vector of size n)
4: Initialize matrix A (random binary matrix of size m × n)
5: Set noise rate $\eta$ (probability of noise bit being 1)
6: Training Process:
7: For epoch=1 to EPOCHS do
8: For batch=1 to BATCH_SIZE do
//Generate LPN Samples
9: Sample noise vector e (each bit is 1 with probability $\eta$)
10: Compute $b = A.s + e \pmod 2$
//Generate Noise Vector
11: Sample noise vector z∼N(0,1) (Gaussian noise)
//Forward Pass: Pass z through $G_\theta$ to generate pseudo-random bits PBS
12: PBS ← $G_\theta(z)$
//Compute Loss: Compute Binary Cross-Entropy (BCE) loss between PBS and b
13: loss ← $bce(PBS, b)$
//Backpropagation
14: Compute gradients of $G_\theta$ with respect to loss
15: Update weights and biases of $G_\theta$ using Adam optimizer
16: end for
17: end for
18: Generate PRNG:
19: Sample noise vector z∼N(0,1)
20: Pass z through $G_\theta$ to generate PBS
//Convert PBS to binary format by thresholding at 0.5
21: $PBS_{binary} \leftarrow (PBS > 0.5)$
22: return $PBS_{binary}$

As shown in Algorithm 1, after training, the generator is used to produce pseudo-random numbers.

### 3.2.2. Entropy-based local density adaptation and deterministic correction

The initial dataset $\mathcal{D}$ consists of binary samples $x_i \in \{0,1\}^{1024}$, generated by the GANLPN. Each sample $x_i$ is a binary vector of length 1024 bits, where the distribution of $0's$ and $1's$ may be imbalanced. To mitigate class imbalance (skewed 0/1 distribution) while retaining spatial coherence, we apply density-dependent bit-flipping as shown in Algorithm 2:
For each bit $x_{ij} = 0$ in sample $x_i$, compute its local neighborhood density $\rho_{ij}$ within a sliding window of size k:

$$\rho_{ij} = \frac{1}{k}\sum_{l=j-\lfloor k/2 \rfloor}^{j+\lfloor k/2 \rfloor} \mathbb{I}(x_{i,l} = 0)$$

Where $\mathbb{I}(.)$ is the indicator function and boundary conditions are handled through reflection padding.
The flipping probability $P_{flip}(x_{i,j})$ is modeled as a function of $\rho_{ij}$, ensuring higher-density regions have a greater likelihood of replacement:

$$P_{flip}(x_{i,j}) = \alpha \cdot (1 - e^{-\beta\rho_{ij}})$$

Where:
$\alpha \in (0,1]$ controls the maximum flipping probability,
$\beta > 0$ adjusts sensitivity to local imbalance.
After entropy-driven flipping, we enforce global balancing via deterministic correction as follows:
Let $\Delta = \left|\sum_{j=1}^{n} x_{i,j} - \frac{n}{2}\right|$ be the residual imbalance. If zeros exceed ones ($\Delta > 0$), flip $\Delta$ uniformly chosen $0 \rightarrow 1$. If ones exceed zeros ($\Delta > 1$), flip $\Delta$ uniformly chosen $1 \rightarrow 0$. This ensures:

$$\sum_{j=1}^{n} x_{i,j} = \left\lfloor \frac{n}{2} \right\rfloor$$

The balanced dataset $\mathcal{D}'$ is reshuffled via a random permutation $\pi$ to eliminate positional bias introduced by bit-flipping: $\mathcal{D}' \leftarrow \pi(\mathcal{D}')$. This ensures uniformity such that the distribution of 0's and 1's is spatially invariant and local patterns are preserved under random reordering.

Algorithm 2: Entropy-based local density adaptation and deterministic correction

Input: Binary sequence $x = (x_1, \ldots, x_n)$ (from GANLPN), Window size k (odd integer, default=5), Hyperparameters $\alpha \in (0,1]$, $\beta > 0$
Output: Balanced and shuffled binary sequence $x' = (x_1, \ldots, x_n)$

// Step 1: Entropy- Based Local Balancing
1: $x_{padded} \leftarrow reflected\_pad(x, k//2)$ // Handle boundaries
2: for j ← 1 to n do

3: if $x_j = 0$ then
//Compute local density of 0s
4: $\rho_j \leftarrow \frac{1}{k}\sum_{i=j-\lfloor k/2 \rfloor}^{j+\lfloor k/2 \rfloor} \mathbb{I}(x_{padded}[i] = 0)$
//Calculate flip probability
5: $P_{flip}[x_j] \leftarrow \alpha \cdot (1 - e^{-\beta\rho_j})$
//Probabilistic flipping
6: if $\text{random}() < P_{flip}[x_j]$ then
7: $x_j \leftarrow 1$
8: end if
9: end if
10: end for
// Step 2: Deterministic Correction
11: zeros ← count(x, 0)
12: ones ← count(x, 1)
13: Compute $\Delta \leftarrow |\text{zeros} - \text{ones}|$
14: if $\Delta > 0$ then
15: if zeros > ones then
// Need to flip $(\Delta/2)$ 0s → 1s
16: $\text{zero}_{indices} \leftarrow \{j \mid x_j = 0\}$
17: $\text{flip}_{indices}$ ← randomly select $\lceil \Delta/2 \rceil$ from $\text{zero}_{indices}$
18: for idx in $\text{flip}_{indices}$ do
19: $x_{idx} \leftarrow 1$
20: end for
21: else
// Need to flip $(\Delta/2)$ 1s → 0s
22: $\text{one}_{indices} \leftarrow \{j \mid x_j = 1\}$
23: $\text{flip}_{indices}$ ← randomly select $\lceil \Delta/2 \rceil$ from $\text{one}_{indices}$
24: for idx in $\text{flip}_{indices}$ do
25: $x_{idx} \leftarrow 0$
26: end for
27: end if
// Step 3: Applies a random permutation π to eliminate positional bias
28: Shuffle the sequence $x' \leftarrow \pi(x)$
29: return $x'$

To guarantee that the final list contains solely prime numbers, which is essential for this work, the binary sequence $x'$ produced by Algorithm 2 is first converted to integers using a binary-to-integer conversion process. A primality test is performed on each integer to verify its primality. The closest prime number is found and used in place of integers that do not satisfy the primality test. The following section describes the steps: converting a binary number to an integer, checking whether a number is prime, and finding the nearest prime number.

### 3.2.3. Binary-to-integer conversion

After Algorithm 2 balancing, the flattened binary sequence undergoes structured transformation to enable primality testing as follows: The 1D array $x' = (x_1, \ldots, x_n)$ is partitioned into n blocks of 1024 bits:

$$X_{samples} \in \{0,1\}^{n \times 1024},\ n = \text{number of samples from the GANLPN}$$

Each block $x_i$ is converted to an unsigned integer:

$$m_i = \sum_{k=0}^{1023} x_i, k \,.\, 2^{1023-k},\ m_i \in [0, 2^{1024} - 1]$$

In this way, the binary number can be reconstructed in decimal format, preserving the statistical properties imparted by the generator. The linear $O(n)$ time complexity to perform this operation (where n is the bit length) ensures that the processing takes a reasonable time even for the large 1024-bit outputs typical of our system. This study adopts Python's native binary-to-integer conversion method to maximize efficiency.

### 3.2.4. Primality testing and nearest prime search

The resulting integer outputs of the binary conversion process are the input dataset for the primality test. These converted numbers (now native integers) are then subjected to rigorous primality testing to determine whether they are prime. The task of questioning whether a natural number is prime is a fundamental problem in number theory and cryptography. The primality test used in this study is the Miller–Rabin primality test, a probabilistic algorithm for determining whether a given number is prime. It is more efficient, especially for large numbers, and is widely used in cryptographic applications. The Miller-Rabin test is based on the properties of strong pseudoprimes. As shown in Algorithm 3, given an odd integer n, the test checks if n is a strong pseudoprime to a randomly chosen base a. If n passes the test for several bases, it is likely prime. The latter is a probabilistic iterative algorithm consisting of rounds. At each round a special parameter $a, 2 \le a \le n - 1$, is chosen called the base of the round.

Algorithm 3: Rubin-Millier primality test

Input: An odd integer $n > 2$ to be tested for primality and a parameter k (number of iterations/witnesses)
Output: " Probably Prime" or "Composite"
// Preliminary Check
1: if $n \le 1$, return "Composite" end if
2: if n is 2 or 3, return "Probably Prime" end if
3: if n is even, return "Composite" end if

4: Write $n-1=2^s.d$, where d is odd
5: Witness Loop (Repeat k times) do
6: Randomly select a base a such that $2 \leq a \leq n-2$
7: Compute $x \leftarrow a^d \bmod n$
8: If $x = 1$ or $x = n-1$ then
9: continue to the next iteration
10: Else
11: Repeat $s-1$ times
12: $x = x^2 \bmod n$
13: If $x = n-1$ then
14: Break and continue to the next iteration
15: end if
16: if $x \neq n-1$ after all repetitions then
17: Return "composite"
18: end if
19: end if
20: end Witness Loop
21: if n passes all k iterations then
22: Return "likely prime"
23: end if

### 3.2.5 Optimized primality test

One significant inefficiency in using the Miller-Rabin algorithm (see Algorithm 3) without optimization is its inability to quickly detect obvious composites before executing the complete modular exponentiation steps. Without prior screening, Miller-Rabin performs full modular exponentiation even on candidates easily divisible by small primes, which can detect many non-prime candidates in milliseconds. As a result, we propose a hybrid primality test that optimizes the Miller-Rabin algorithm (see Algorithm 4) by leveraging the efficiency of divisibility. The main point is that the majority of composite numbers can be identified as non-prime in short order by testing for divisibility by a predetermined list of small prime numbers, thereby eliminating most composites before using the Miller-Rabin procedure. Simple, small-prime divisibility tests drastically reduce the number of candidates that need to be tested. This hybrid approach offers an optimal balance between computational efficiency and accuracy, minimizing the number of Miller-Rabin iterations while preserving correctness.
Definition:
Let S be the set of small primes, defined as:

$$S = \{p \in \mathbb{P} | 1 < p < 100\}$$

Where $\mathbb{P}$ denotes the set of all prime numbers.
A positive integer n is a composite if and only if the following condition holds:

$\exists p \in S$ such that $p \mid n$

However, composite numbers $n_\theta \in \mathbb{Z}^+$, in the form $n_\theta = p \cdot q$, where p and q are prime such that, $p, q \notin S$ could evade this heuristic and be falsely considered prime. To avoid this, we supplement the method with the Miller-Rabin primality test to give us correctness and efficiency. With the addition of the Miller-Rabin primality test, we focus only on the composite of the form $n_\theta$ and $\mathbb{P}_x \in \{\mathbb{P} \setminus S\}$, where $\mathbb{P}_x$ is the set of all prime numbers excluding the small primes in S. Refer to Algorithm 4 for the optimized primality test.

Algorithm 4: Optimized primality test
Input: An odd integer $n > 2$ to be tested for primality and a parameter k (number of iterations/witnesses)
Output: "Likely Prime" or "Composite"
// Preliminary Check
1: if $n \leq 1$, return "Composite" end if
2: for each element p in S do
3: if n mod p == 0 then
4: if n == p then
5: return "Likely Prime"
6: else
7: return "Composite"
8: end if
9: end if
10: end for
11: Write $n-1=2^s.d$, where d is odd
12: Witness Loop (Repeat k times) do
13: randomly select a base a such that $2 \leq a \leq n-2$
14: compute $x \leftarrow a^d \bmod n$
15: if $x = 1$ or $x = n-1$ then
16: continue to the next iteration
17: else
18: repeat $s-1$ times
19: $x = x^2 \bmod n$
20: if $x = n-1$ then
21: break and continue to the next iteration
22: end if
23: if $x \neq n-1$ after all repetitions then
24: return "composite"
25: end if
26: end if

```
27: End Witness Loop
28: if n passes all k iterations then
29:           return “likely prime”
30: end if
```

As shown in Algorithm 4, lines 2-10 merge our Definition with the Miller-Rabin primality test to quickly identify small primes, composite numbers, and multiples of small primes, as defined in this study. Checking divisibility by small primes in S is much faster than running the full Miller-Rabin test. If n is a multiple of these small primes in S, we can immediately determine whether n is prime or composite without proceeding to the Miller-Rabin steps. This step eliminates many composite numbers early in the process. In practice, many numbers tested for primality are either small primes or composite numbers, multiples of small primes. This optimization avoids additional computation for such cases.

#### 3.2.6. Nearest prime search

The system launched an optimized nearest-prime search for integers that failed primality checks. The study merged probabilistic primality testing with incremental search (see Algorithm 5). The probabilistic primality-testing method with incremental search is an appropriate alternative for determining the nearest prime to an integer, especially in this study, which involves large integers.

Algorithm 5: Nearest prime search

```
Input: A list of numbers (numlist)     // Converted binary to integer samples
Output: A list of primes (primelist)

1: initialize primelist as an empty list.
2: for each number s in numlist do:
3:       set n = s
4:       if n is divisible by 2 then
5:           increase n by 1   // to make it odd
6:       end if
7:     while true do
8:          if n is prime then    // using the Algorithm 4
9:              append n to primelist
10:               break the loop
11:          else
12:              increase n by 2   // to skip even numbers
13:          end if
14:      end while
15: end for
16: return primelist
```

As shown in Algorithm 5, for even inputs greater than 2, it increments n by 1 to start testing at the following odd number, since all larger even numbers are composite (lines 4 - 6). The core loop then checks each subsequent odd candidate using Algorithm 4 (with k rounds for probabilistic certainty) and returns the first number that passes (lines 7 – 14). The algorithm cuts the search space in half, doubling its speed compared to checking all odd numbers, making it ideal for large primes.

## 4. Results and Discussion

This section gives a thorough explanation of the experiments conducted to evaluate the performance and reliability of the proposed GANLPN-based approach. All simulations were implemented in Python 3.12.7 within the Jupyter Notebook framework of the local work-station. The specifications of the machine running the experiments were an AMD Athlon Silver 3050U processor with Radeon Graphics running at a base frequency of 2.30 GHz, a 64-bit Windows 10, and 20 GB of RAM. To test the quality of the generated pseudorandom outputs, the statistical test suite NIST SP800-22 was used.

### 4.1. Hyperparameters and model configuration

To rigorously evaluate the effectiveness of our GANLPN PRNG, we conducted extensive testing and carefully fine-tuned the network’s hyperparameters, as detailed below:

The generator takes a 256-dimensional input vector sampled from a Gaussian distribution, serving as the initial noise seed, and produces 1024-bit outputs through a series of fully connected layers with dimensions 512, 1024, 2048, and 1024. A ReLU activation function between layers introduces nonlinearity, helping the model learn complex patterns. The final layer uses a sigmoid activation, which scales all outputs to the range [0,1], allowing them to be interpreted as probabilities before thresholding into binary bits.

The model processes data in batches of 32 samples per iteration during the training phase, aiming to optimize performance metrics for both computational efficiency and gradient stability. The computational efficiency-oriented framework uses the Adam optimizer for training, a common choice for DL work that adapts learning rates optimally across weights, with a learning rate of 0.0002 to facilitate non-rash weight updates. The loss function, or loss criterion, is binary cross-entropy (BCELoss), which measures the divergence between the generator's output and the set of target LPN samples. The goal of the model is to produce bit sequences that are statistically indistinguishable from those produced using LPN. Training runs for 1000 epochs, with periodic loss evaluations every 100 epochs to monitor convergence. The LPN part of the PRNG is responsible for its cryptographic security. The LPN operates using a secret key s of length n=256 bits fixed during operation. The noise rate $\eta$ is set to 0.1, which means that approximately 10% of the bits are flipped in the samples generated. This feeds computational difficulty to prevent easy recovery of the key. Each LPN sample has a size of 1024 bits, the same size as the output width of the generator. The main security parameters, n (secret size) and $\eta$ (noise rate), may be adjusted to enhance security: a larger n improves resilience to attacks, and a larger $\eta$ improves unpredictability (but also increases training difficulty). The sample width (1024 bits) is flexible and can be modified based on application needs.

Our implementation of the GANLPN model follows the configuration outlined in this section. While adversarial training time is usually lengthy due to the need for a GAN architecture, we achieved about 6 minutes of training time with our optimized setup on a standard CPU-based platform. We have demonstrated a 100% training success rate for the GANLPN model on 60 runs, with all 6 minutes of converged results being consistent. As a result, the model has exceptional stability and requires minimal setup time, making it highly reproducible and cost-effective when operated in a resource-constrained environment. Although there will be a slight increase in the time needed to train the GANLPN model compared to traditional PRNG models, all computations are quick enough to make the increased training time acceptable for both research and deployment. Once trained, the model's inference phase is efficient and can generate large batches of pseudorandom samples with significantly lower computational overhead than during the initial training stage.

## 4.2. Analysis of the proposed GANLPN PRNG experimental results

Per standard PRNG requirements, the system must exhibit key properties, including expansibility, unpredictability, and pseudorandomness. The following breakdown heightens on these characteristics:

Expansibility: The requirement that the PRNG's output length exceed its input length. In the proposed design, we start with a 256-bit seed and expand it to 1024 pseudorandom bits, a fourfold expansion. We achieve this using several neural layers that go from 256 bits to 512, then to 1024, up to 2048, and back down to 1024. This transformation makes it more challenging to guess the seed from the output. The nonlinearity introduced by this transformation, associated with the LPN problem, increases computational complexity, thereby improving resistance to seed inversion.

Unpredictability: This property implies that, based solely on a sequence generated by a generator, it is impossible to determine the internal state of the generator or future outputs. Unpredictability has two aspects: forward security and backward security. Forward security ensures that knowing the current output does not divulge any information about any previous outputs. On the other hand, backwards security ensures that even if you observe the current production or state, it won't affect what comes next. These protections help keep the PRNG secure.

The GANLPN model achieves robust unpredictability, ensuring both forward and backward security through a combination of cryptographic hardness and neural network design. Forward security is ensured by the difficulty of solving the LPN problem, where each output block of 1024 bits is generated using a random 256-bit secret key and fresh noise ($\eta$=0.1), so that even if the current outputs are compromised, it is impossible to guess future outputs. It is provably hard to solve LPN (reducing to the worst-case lattice problem); hence, even if an attacker could use the first few outputs, it would still be too costly for the attacker to predict future output bits. The generator's one-way function design (256→512→1024→2048→1024) prevents inversion attacks due to the complex, nonlinear ReLU activation and the high-dimensional transformation. Each output is generated independently from a new 256-bit noise vector, discarding the input noise after use, preventing the reconstruction of past outputs. The asymmetry of the LPN problem (requiring the solution of noisy linear equations) and the neural network's obfuscation both defend against forward prediction and backward tracing, and the 256-bit key and noise injection provide resistance to brute-force and algebraic attacks.

Pseudorandomness: The GANLPN-based PRNG transforms data from a low-dimensional space into a high-dimensional output uniformly distributed. It passed the NIST SP800-22 statistical tests, showing it has strong pseudorandom traits and behaves well statistically.

### 4.2.1. NIST SP800-22 tests

The SP800-22 statistical test suite, developed by the National Institute of Standards and Technology (NIST), is a known criterion for checking the quality of PRNGs [37]. The suite consists of 15 main test categories (and 188 subtests) that each test a different statistical property of binary sequences. Each test provides a p-value that indicates whether the analyzed sequence exhibits behavior consistent with a source of true randomness. As per the SP800-22 guidelines, a sequence passes a test if its p-value is greater than a predefined significance level, $\alpha$ (usually an $\alpha$ between 0.001 and 0.01), which indicates a high probability (with a confidence of $1-\alpha$) that the sequence behaves statistically random. These criteria ensure the robustness of the test results and the credibility of any claims regarding the randomness of a PRNG's output.

The evaluation of a sequence under the SP800-22 standard is based on two key conditions for test success:

- The rate of sequences in passing the test thresholds: To assess the statistical validity of a sequence, the p-value for each sequence must not fall below the chosen significance level, $\alpha$. The proportion of sequences passing a given test is then compared against an acceptable threshold, calculated using a confidence interval defined as $\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$, where $\hat{p} = 1-\alpha$, and m is the number of tested sequences. If the observed pass rate lies outside this interval, the sequence set may be considered non-random. For instance, with $\alpha = 0.01$ and m = 1000, the acceptable pass rate range is $\left(0.99 \pm 3\sqrt{\frac{0.99(0.01)}{1000}}\right) = (0.9805607, 0.9994392)$, and any pass rate below 0.9805607 suggests insufficient randomness.
- The distribution of P-values to check for uniformity: To verify the uniformity of P-values, the interval [0, 1] is partitioned into 10 equal sub-intervals, and the frequency of P-values within each bin is recorded. A chi-squared ($\chi^2$) goodness-of-fit test is then employed to evaluate whether the observed distribution aligns with the expected uniform distribution. The test statistic is calculated as $\chi^2 = \sum_{i=1}^{10} \frac{(F_i - E)^2}{s/10}$, where $F_i$ represents the observed count of P-values in the $i^{th}$ sub-interval, and $E = s/10$ is the expected count under uniformity for a sample size s. The resulting P-value is derived such that $P-value_T = igamc\left(9/2, \chi^2/2\right)$. A $P-value_T \geq 0.0001$ suggests that the P-values are uniformly distributed. To ensure statistical reliability, a minimum sample size of 55 sequences is required for the analysis.

We started with 1,024,000,000 binary sequences from the proposed GANLPN PRNG as NumPy arrays, then changed them to strings to enable efficient bit-level operations while preserving spatial relationships between bits, as it allowed Algorithm 2 to compute local bit densities ($\rho_j$) using sliding-window operations. By probabilistically flipping bits according to $P_{flip}[x_j] \leftarrow \alpha \cdot \left(1 - e^{-\beta\rho_j}\right)$, the method effectively mitigated clustered bit patterns, a known vulnerability in raw GAN-LPN outputs, while retaining global statistical properties. This operation improves the NIST test pass rates. The SP800-22 test suite required a sequence length of at least 1 million bits to determine whether it passed as a test of true randomness. After running Algorithm 2 on the generated bit sequence to improve randomness, the total bitstream data of 1,024,000,000 bits, we split the data for testing by the SP800-22 test requirement into 102 complete segments, each

1,003,921 bits in length. We executed the evaluation test using the default testing options when using the SP800-22 test suite for consistency and reproducibility of test results.
As shown in Table 3, all of the statistical tests, except for the Random Excursion (Variation) test, passed at least 97 out of 102 samples of the binary sequences. The minimum pass rate of the Random Excursion (Variation) among 55 binary sequences tested was approximately 52. P-values for the pass tests were all significantly above 0.01, and the observed pass rates were all well above the required qualification mark. These results, in conjunction, confirmed that the artillery of tests proved the statistical randomness of the generated sequences. The results further demonstrated the acceptance of the proposed GANLPN-based PRNG under the NIST SP800-22 randomness tests.

**Table 3:** SP800-22 Test Results

| No | Test | P-value | Proportion | Passed/failed |
|---|---|---|---|---|
| 1 | Frequency | 0.494392 | 101/102 | Passed |
| 2 | Block Frequency | 0.798139 | 101/102 | Passed |
| 3 | Cumulative Sums | 0.401199 | 102/102 | Passed |
| 4 | Runs | 0.062821 | 102/102 | Passed |
| 5 | Longest Run | 0.534146 | 102/102 | Passed |
| 6 | Rank | 0.023545 | 99/102 | Passed |
| 7 | Spectral Discrete Fourier Transform | 0.699313 | 101/102 | Passed |
| 8 | Non-overlapping template matching | 0.014550 | 100 /102 | Passed |
| 9 | Overlapping template matching | 0.419021 | 102/102 | Passed |
| 10 | Maurer's universal statistical test | 0.699313 | 100/102 | Passed |
| 11 | Approximate entropy test | 0.262249 | 99/102 | Passed |
| 12 | Random excursions test | 0.202268 | 52/55 | Passed |
| 13 | Random excursions variant | 0.019188 | 53/55 | Passed |
| 14 | Serial test | 0.090936 | 102/102 | Passed |
| 15 | Linear Complexity | 0.474986 | 99/102 | Passed |

#### 4.2.2. Comparative analysis of GAN-based PRNGs

Table 4 presents a comparative analysis of existing GAN-based PRNG schemes.
A secure PRNG is expected to have an input space of at least 128 bits to resist brute-force attacks. The input sizes of [26], [27] and [24] are limited to 32, 64, and 36 bits, respectively. Only the schemes of [28], [29] met the suggested security benchmark (more than 128 bits). Inference time measures the time it takes the trained generator to produce random values for a given input seed, which reflects the system's responsiveness. Throughput gives the amount of random data generated in a given time frame. The experimental evaluation shows that the proposed GAN-LPN PRNG achieves superior performance, recording the shortest inference time (181 ms) and highest throughput (0.071 GB/s) among the compared methods. Notably, this boost in performance doesn't sacrifice the quality or randomness of the results, as shown by passing the NIST SP800-22 statistical tests (see Table 3).
While the work by Wu et al. [29] passed all 15 NIST statistical randomness tests, they acknowledged that their designs provided no inherent cryptographic security. A next step could be combining the computational hardness of the LPN problem with neural networks, as a possible path toward constructing a cryptographically secure PRNG. Based on this recommendation, in the current work, we have successfully designed a GAN with LPN hardness to construct a PRNG with statistically random outputs and cryptographic sturdiness.

**Table 4:** Efficiency Comparison with other PRNGs Based on GAN

| Authors | Core security mechanism | Input size | Random bits sequence tested | Inference time | Throughput | NIST pass test results |
|---|---|---|---|---|---|---|
| Proposed Scheme | GAN with LPN | 256 | 1,073,152 | 181 ms | 0.071 GB/s | Yes |
| Wu et at. [29] | GAN with GA | 256 | 1,048,576 | 276 ms | 0.81 GB/s | Yes |
| Okada [27] | WGAN | 36 | 1,048,576 | 252 ms | 0.32 GB/s | Yes |
| Bernardi et al. [24] | GAN | 32 | 1,000,000 | 186 ms | 0.23 GB/s | No |
| Wu [28] | GAN | 256 | 1,048,576 | 213 ms | 0.95 GB/s | Yes |
| Kim [26] | GAN | 64 | 1,099,200 | 142 ms | 1.2 GB/s | No |

### 4.3. The GANLPN prime dataset generation

After completing end-to-end training and validation of the GANLPN generator, the system can produce cryptographically secure pseudorandom numbers at scale. For the prime dataset generation, we first generated 1,115,000 pseudorandom 1024-bit numbers using the GANLPN framework. These numbers were initially represented as multidimensional arrays, which we flattened into a continuous 1D binary sequence of 1,141,760,000 bits (1115000 × 1024) to facilitate the Algorithm 2 bit-balancing process, which eliminates distributional bias (see Algorithm 2).
The balanced output was then reshaped back into 1M blocks of 1024 bits and converted to unsigned integers via the operation $m_i = \sum_{k=0}^{1023} x_i, k \,.\, 2^{1023-k}$ where $m_i \in [0, 2^{1024} - 1]$, implemented efficiently through Python's int(".join(map(str, sample)), 2) for exact binary-to-integer mapping. After mapping the binary data to integers, we optimized the next step in primality analysis and the nearest-prime search phase by grouping 1,115,000 samples into 223 batches of 5,000 integers. Breaking the process into batches facilitated parallelism, thereby optimizing memory availability. For each batch, we performed two operations: (1) Optimized Miller-Rabin primality testing with 40 iterations ($k = 40$) to achieve a false positive probability of $< 2^{-80}$, and (2) for composite numbers, a probabilistic nearest-prime identification with incremental search (see Algorithm 5). On average, each batch took 50 minutes, resulting in an anticipated total runtime of about 185 hours for all 1,115,000 samples. This length of time is directly related to large primes:

- Prime Gap Scaling: For 1024-bit numbers ($n \approx 2^{1024}$), the average prime gap grows logarithmically as $\ln(n) \approx 710$. This means that when searching for the nearest prime to a composite 1024-bit number, the algorithm must test approximately 355 candidates on average (checking both above and below the target value). Each candidate requires a complete test of Algorithm 5, which involves modular exponentiation. Since modular exponentiation on b-bit integers hold a time complexity of $O(b^3)$, the total cost of the nearest-prime search becomes $O(b^3 \cdot \ln(n))$ per composite number. This cubic scaling makes primality testing dramatically slower for 1024-bit integers compared to smaller numbers, explaining the significant runtime observed in the study.
- Iterative Testing Overhead: Algorithm 4 achieves cryptographic certainty through iteration: each round reduces the false positive probability by a factor of 4. With 40 iterations, the test reaches a false positive rate of $< 2^{-80}$, which is considered negligible for

cryptographic applications. However, this robustness comes at a linear runtime cost. Each iteration performs full modular exponentiation ($O(b^3)$), so 40 iterations require 40 times more work than a single test. While fewer iterations (e.g., $k = 20$) would halve the runtime, they would also increase the false positive probability to $< 2^{-40}$, which is unacceptable for generating primes used in secure systems like RSA. Thus, the 40-iteration overhead is a deliberate design choice that reflects the trade-off between computational efficiency and cryptographic security.

The 50-minute batch run time is in line with our expectations, primarily due to the time required to search for the next prime number; its complexity increases with the number of composite numbers. The time observed is a demonstrable trade-off needed to meet the security requirements for generating prime numbers. The complete dataset of the generated primes can be found at the link: https://drive.google.com/drive/folders/1-LGuhGb7rN-MsxVp_KH2yPWdEUHN673y?usp=sharing

## 4.4. GANLPN security computational hardness

Suppose there exists a polynomial-time adversary $\mathcal{A}$ that can distinguish the output of our GANLPN generator from a uniform random distribution with non-negligible advantage $\epsilon$. We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to solve the LPN problem. $\mathcal{B}$ is given an LPN instance $(A, b)$, where $b = sA + e$, and must decide if this is a valid LPN sample or uniform random. Assume that there is an adversary $\mathcal{A}$ that runs in polynomial time and can distinguish the output of our GANLPN generator from a uniform random distribution with non-negligible advantage $\epsilon$. We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to solve the LPN problem. $\mathcal{B}$ is given an LPN instance $(A, b)$, where $b = sA + e$, and must decide if this is a valid LPN sample or a uniform random.

$\mathcal{B}$ proceeds as follows:

- $\mathcal{B}$ feeds the LPN sample $(A, b)$ to the GANLPN generator's discriminator (or uses it to simulate a generator output) and then presents the result to the adversary $\mathcal{A}$.
- $\mathcal{B}$ runs $\mathcal{A}$ on this simulated output.
- If $\mathcal{A}$ outputs that the sample came from GANLPN, $\mathcal{B}$ guesses that $(A, b)$ was a valid LPN sample. If $\mathcal{A}$ outputs random, $\mathcal{B}$ guesses that $(A, b)$ was uniformly random.

The essential point is that the success probability for $\mathcal{B}$ depends on how much of a distinguishing advantage $\mathcal{A}$ has in distinguishing GANLPN outputs from random outputs. If the instance is a real LPN instance, then the adversary $\mathcal{A}$ receives an output that appears to be a valid GANLPN output. If the instance is random, then the adversary $\mathcal{A}$ gets an output that looks random. Therefore, $\mathcal{A}$'s ability to distinguish GANLPN gives $\mathcal{B}$ the ability to distinguish LPN from random, which allows $\mathcal{B}$ to solve the LPN problem. Since we assumed LPN is hard, no such efficient $\mathcal{A}$ can exist, and therefore, our GANLPN construction is secure.

We explicitly acknowledge that GANLPN provides heuristic security rather than formal cryptographic proofs. The security of GANLPN rests on a computational hardness argument rather than a formal reduction proof. Our approach assumes that if the GAN successfully learns the distribution of LPN samples, then distinguishing GANLPN outputs reduces to solving the LPN problem. GANLPN provides empirical security based on observed resistance to statistical and cryptographic attacks, analogous to heuristic constructions in post-quantum cryptography.

Neural networks are universal function approximators that learn statistical distributions but cannot guarantee exact mathematical properties. GANs' internal representations are not easily interpretable using traditional cryptographic methods. While it is possible to formally verify each step of an algorithm through proof of correctness, it is challenging to provide reductionist evidence of the security of neural networks, due to their nature as black-box approximators. As such, GANLPN follows the methodology outlined in NIST SP 800-22 through empirical validation rather than proof.

## 4.5. GANLPN security comparison with standard PRNG constructions

We contextualize the security of GANLPN by comparing it to two classic PRNG constructions: the BBS generator and the AES-based CTR DRBG (see Table 5).

**Table 5:** Security Comparison with Standard PRNG Constructions

| Feature | GANLPN | BBS | AES-CTR DRBG |
|---|---|---|---|
| Security assumption | Leans on the hardness of the LPN problem | Depends upon the hardness of the Integer Factorization problem | Relies on the security of the AES block cipher as a pseudorandom permutation |
| Post-Quantum security potential | The LPN problem is thought to withstand attacks by quantum computers, and thus GANLPN can be considered as a potential participant in post-quantum cryptography | The security is rooted in factoring, and factoring can be broken efficiently using Shor's algorithm, given a powerful enough quantum computer | While not completely broken, Grover's quantum algorithm squares the search time, effectively halving the security level of AES (AES-128 would have ~64 bits of post-quantum security) |
| Nature of security proof | A reduction to the LPN problem under the assumption that the GAN learns the target distribution. This is a novel and less explored security argument | A reduction to the Quadratic Residuosity problem | Has a reduction to the PRP security of AES, which is well-established but not based on a number-theoretic hard problem |
| Resistance to known attacks | No known attacks. Its security is enhanced by its direct dependence on the well-studied LPN problem. A current lack of known attacks provides positive initial evidence of security | It is resilient against known attacks, provided a suitable choice of prime moduli is made | Many standard implementations of CTR-DRBG still rely on table-based AES and thus are susceptible to cache side-channel attacks |

As shown in Table 5, GANLPN explores a unique point in the design space. It offers a promising path to post-quantum security based on a learning-based paradigm, and its direct foundation in the LPN problem provides a strong, credible security base. The initial absence of any known attacks is a positive and encouraging result. Its contribution is foundational, opening a new avenue for constructing pseudorandom generators from hard learning problems.

## 4.6. Analysis of GANLPN generated prime quality

To evaluate the cryptographic quality of primes generated by our GANLPN method, we conducted quality testing across multiple residue classes using the dataset of 1,115,000 generated primes. We also conducted testing against known factorization attacks and performed a

vulnerability analysis. As summarized in Table 8, the GANLPN method achieved a 100% success rate across eight critical tests, demonstrating that GAN-generated primes meet the standards for generating cryptographic primes.

**Table 6:** Cryptographic Uniformity Test

| Distribution Modulo 30 | | | | Last Digit Distribution | | |
|---|---|---|---|---|---|---|
| Residue | Count | Expected | Deviation | Last digit | Count | Percentage |
| 1 | 139,084 | 139375.00 | 0.21% | 1 | 278,865 | 25.01 |
| 7 | 139,058 | 139375.00 | 0.23% | 3 | 278,581 | 24.98 |
| 11 | 139,781 | 139375.00 | 0.29% | 7 | 278,692 | 24.99 |
| 13 | 139,420 | 139375.00 | 0.03% | 9 | 278,862 | 25.01 |
| 17 | 139,634 | 139375.00 | 0.19% | | | |
| 19 | 139,712 | 139375.00 | 0.24% | | | |
| 23 | 139,161 | 139375.00 | 0.15% | | | |
| 29 | 139,150 | 139375.00 | 0.16% | | | |

**Table 7:** Factorization Attacks Resistance

| Pollard's p-1 Attack Resistance | | Smoothness Vulnerability Analysis | |
|---|---|---|---|
| Metric | Value | Metric | Value |
| Primes Analyzed | 1,115,000 | Primes Analyzed for Smoothness | 1,115,000 |
| Resistant to Pollard's p-1 | 1,115,000 | Average p-1 Smoothness | 0.75% |
| Resistance Percentage | 100.00% | Average p+1 Smoothness | 0.75% |
| Cryptographic Standard | >95% | Potentially Vulnerable Primes | 0 |
| Safe Primes Found | 1,013 | Vulnerability Percentage | 0.00% |
| Strong Primes Found | 1,115,000 | Acceptable Threshold | <5% |

**Table 8:** Cryptographic and Statistical Test Summary

| Test type | GANLPN Result | Standard | Passed | Significance |
|---|---|---|---|---|
| Gap Distribution | Ratio: 1.003 | 0.9-1.1 | YES | Critical |
| Modulo 30 Uniformity | p=0.719 | p>0.05 | YES | Critical |
| Multi-Modulus Test | All p>0.05 | All p>0.05 | YES | Important |
| Last Digit Distribution | p=0.976 | p>0.05 | YES | Important |
| Pollard p-1 Resistance | 100.0% | >95% | YES | Critical |
| Smoothness Vulnerability | 0.00% | <5% | YES | Critical |
| Bit Length Consistency | 99.8% | >95% | YES | Important |
| Safe Prime Generation | 0.09% | Natural | YES | Validation |

### 4.6.1. Uniformity distribution analysis

As shown in Table 8, we researched the distribution modulo 30, which is the most critical of all the tests for prime uniformity. Primes larger than 30 are asymptotically evenly distributed among the eight admissible residue classes. Our results showed a practically perfect fit with theoretical predictions, with a chi-squared statistic of $\chi^2 = 4.5137$ ($p = 0.719$), which suggests there was no significant deviation from uniformity. The largest deviation from what was expected was only 0.29%, and all eight residue classes showed a deviation below 0.3%. This indicates that our generation method successfully captures the fundamental distribution properties of primes.

The count of the last digits serves as a delicate check for low-order bit biases, because primes greater than 10 should be uniformly distributed among the digits {1, 3, 7, 9}. We looked at 1,115,000 primes, and we observed a nearly perfectly uniform distribution (Table). The chi-squared test indicated perfect uniformity ($\chi^2 = 0.2070$, $p = 0.976$) and even with the highest (or least) 0.02% deviation from the theoretical expectation of 25% in any of the four digits. A distribution that indicated an absence of any systematic patterns in the least significant bits, which is an essential requirement for cryptographic security.

The alignment in results of all tests with a maximum deviation of no greater than 0.5% and p-values above 0.05 attributes valid empirical proof of the cryptographic quality of the generated primes. The naturally occurring safe primes (0.09%) in the generated data support further security proof. They represent the gold standard against specialized factorization attacks. The biosynthesis of safe primes in the GANLPN output markedly demonstrates the model's ability to glean deep mathematical structures beyond the surface-level characteristics of primes.

With an average bit length consistency of 99.8%, all generated primes are guaranteed to have the same cryptographic strength. Hence, there are no concerns regarding weak keys due to only slight variations in the sizes of the primes. The gap distribution ratio of 1.003 means that the average distance between successive primes has mathematical expectations that closely correlate (710.13 compared to the theoretical value of 710).

### 4.6.2. Pollard's p-1 attack and smoothness vulnerability analysis

Pollard's p-1 method of factorization uses primes where p-1 can be composed mainly of small factors. We have examined all 1,115,000 primes that we generated and found that there was perfect resistance to this algorithm, since in p-1, 100% of the primes have large prime factors. This complete resistance to this method of factorization is well above the minimum of 95% defined for cryptographic applications, providing strong security against one of the more practical methods of factorizing improperly generated primes. We also found 1,013 safe primes (primes of the form $p = 2q + 1$ where q is prime). Safe primes offer the maximum resistance to Pollard's p-1 attack and are particularly desirable in the context of Diffie-Hellman key exchange and other protocols that rely on discrete logarithms.

Smoothness-based attacks focus on primes in which p-1 or p+1 primarily contain small factors. We identified zero primes as potentially vulnerable to attack with a smoothness of 0.00% (0.75% for both p-1 and p+1). We find this notable, as it indicates nearly perfect resistance against specialized factorization attacks. Notably, this property was not a result of explicit constraints enforced during the training process to enhance cryptographic strength.

### 4.6.3. Comparison of prime generation methods

To assess bulk prime generation approaches, the existing prime number generation methods were tested in terms of runtime, uniformity, and security against the proposed GANLPN to validate their cryptographic application. To ensure a fair comparison, we timed each method for 60 seconds and repeated this process 100 times to confirm the results, which are in Table 9.

**Table 9:** Comparison of GANLPN with Existing Prime Generation Methods

| Method | No. of primes | Runtime (p/s) | Uniformity(p-value) | | | Security | | |
|---|---|---|---|---|---|---|---|---|
| | | | Mod 30 | Last Digit | Expected | Pollard Attack (%) | Smoothness (%) | Bit Consistency (%) |
| OpenSSL | 926 | 15.42 | 0.0431 | 0.0121 | > 0.05 | 100 | 0 | 100 |
| Trial Division | 93 | 1.55 | 0.8486 | 0.9131 | > 0.05 | 100 | 0 | 74 |
| Hybrid Sieve | 86 | 1.43 | 0.4433 | 0.1791 | > 0.05 | 100 | 0 | 72 |
| Miller-Rabin-64 | 19 | 0.32 | 0.4288 | 0.5221 | > 0.05 | 100 | 0 | 71 |
| Sequential Search | 25 | 0.42 | 0.9948 | 0.8000 | > 0.05 | 100 | 0 | 75 |
| Fermat with MR | 23 | 0.38 | 0.9090 | 0.8232 | > 0.05 | 100 | 0 | 68 |
| Gordon Strong Prime | 0 | 0.00 | N/A | N/A | > 0.05 | N/A | N/A | N/A |
| Safe Primes | 0 | 0.00 | N/A | N/A | > 0.05 | N/A | N/A | N/A |
| GANLPN (proposed) | 121 | 2.02 | 0.9980 | 0.9541 | > 0.05 | 100 | 0 | 100 |

The results unequivocally support the statement that the existing methods are not efficient for generating large numbers of primes for cryptographic use. This conclusion is drawn from a multi-faceted analysis of the results (see Table 9) as follows:

The performance chasm between the industry-standard OpenSSL (926 primes, 15.42 p/s) and the conventional methods is profound. All the examined conventional bulk prime generation approaches, except for OpenSSL, produced fewer than 100 primes within the given time. The Gordon Strong Prime and Safe Prime techniques failed to find any primes within the allowed time (0 p/s), indicating a notable limitation for bulk prime generation. Although these methods aim for strong security in theory, their high computational cost makes them impractical and unsuitable for large-scale prime generation.

While OpenSSL provides the necessary efficiency for generating bulk, throughput performance creates a quality trade-off. With its 15.42 p/s throughput rate, OpenSSL is the only tested method that can come close to being bulk suitable in terms of throughput. However, the noticeable drawback is that OpenSSL fails the statistical uniformity test. The p-values of 0.0431 and 0.0121 suggest that OpenSSL optimization for speed can introduce a degree of predictability into the randomness of the output.

In this context, the performance of the proposed GANLPN is particularly significant. It generates 121 primes at a rate of 2.02 primes per second. While this is approximately 7.6 times slower than OpenSSL, it is 1.3 to 6.3 times faster than the other conventional methods. More importantly, it achieves this while maintaining perfect security scores and near-perfect statistical uniformity. This defines the proposed approach as a potential generator that is efficient for practical cryptographic usage.

The proposed prime generation method establishes a new benchmark for balanced performance, positioning itself as a superior alternative to the conventional methods and a more robust one than OpenSSL in terms of quality for bulk prime generation.

## 4.7. Ethical and misuse of the GANLPN dataset

The primary and approved purpose of this dataset is to further defensive cryptographic strengths by implementing anomaly detection on weak random number generators, conducting an academic study of the number-theoretic properties of primes, and strengthening cryptographic standards by identifying and understanding potential weaknesses in generation algorithms. The generated primes are not certified for production cryptographic use, such as key generation or security-critical systems. We specifically restrict the dataset from use in any offensive security research or actions that subvert cryptographic systems. The dataset should be used exclusively for research, educational, and benchmarking purposes. While openness may create opportunities for a rational adversarial approach to exploitation, ultimately, the development of the security community in a more transparent, defensive-cryptography environment is more valuable than exploiting those opportunities for an adversarial security approach.

## 4.8. Future scope of the study

To better understand primes in the context of cryptography, researchers can use ML to examine patterns in prime gap distributions. Kernel density estimation (KDE) may be a useful visualization or clustering approach to show how, and whether, prime gaps are distributed, and whether they have significant gaps that do not follow the underlying random distribution. DL methods designed to detect sequential patterns can help identify temporal patterns. Researchers can train a bidirectional Long Short-Term Memory network with attention mechanisms on sequences of prime gaps to identify learnable patterns, using windowed inputs of 10-20 consecutive gaps to capture local dependencies. Future researchers could explore quantum-enhanced prime-search algorithms to accelerate the identification of primes with significant gaps. One task is to transform the nearest-prime search into a quadratic unconstrained binary optimization problem, a type of problem that quantum annealers, such as D-Wave, can solve. Researchers could first validate this approach on shorter bit lengths (e.g., 128-bit primes) before scaling it to 1024-bit primes.

It is important to extend the GANLPN pipeline to a broader range of primes (2048-bit and larger). The modifications require architectural adjustments (generator redesign) to achieve gradient stability during training. Principal tasks include increasing the entropy-balancing window size (k) to accommodate larger prime sizes and measuring compliance with NIST SP 800-22. Comparative runtime improvements and adherence to theoretical prime-density expectations indicate success.

One promising research direction involves training DL models to infer the structural properties of RSA keys, particularly using datasets of 1024-bit prime numbers. The aim is to determine whether neural networks can identify hidden patterns or statistical features in these large

primes that may help reconstruct a part of the RSA private key, such as a subset of bits of the primes or the private exponent. Success in this area suggests that RSA key-generation processes may exhibit detectable regularities, potentially weakening the assumed unpredictability of RSA.

The GANLPN hybrid architecture, formally security proof, is meant to encourage researchers to measure how the generator's 4:1 expansion preserves entropy and whether its nonlinear transformations preserve the original LPN noise distribution ($\eta$=0.1). Such a task involves showing that the generator's outputs are computationally indistinguishable from true LPN samples, even with the neuron approximations, possibly through reductionist arguments that show breaking GANLPN implies solving the complex LPN problem.

## 5. Conclusion

This study introduces GANLPN, a generative adversarial network combined with the learning parity with noise problem, to produce balanced, high-entropy 1024-bit primes as a solution to bulk prime generation and the unavailability of large prime datasets, while also showing promise for future research on prime gap distributions and patterns. The GANLPN framework offers a novel combination of neural networks that provide strong pseudorandomness, leveraging the difficulty of the LPN problem, and a GAN for generating output much faster through adversarial training. Compared to other GAN-based PRNGs, the proposed GANLPN provides better theoretical security guarantees (forward and backward security) and strong alignment with cryptographic standards, as demonstrated by its robust performance in the NIST SP 800-22 tests. The public release of our 1,115,000 1024-bit prime dataset opens multiple avenues for advancing both cryptographic construction and cryptanalytic research. To fully utilize this publicly available prime dataset, researchers should conduct the following key next steps: (1) Use the dataset to evaluate existing cryptographic deployments for weak primes while establishing improved generation defaults to resist classical and quantum attacks. (2) Train ML models to identify weak prime patterns and distributions that threaten the security of keys. (3) Accelerate Pollard's p-1 and Elliptic Curve factorization methods by identifying smooth primes. Finally, we want to foster collaborations that will extend this dataset with specialized domains of prime classes, while exploring novel mechanisms to provide ethical safeguards against misuse.

## References


[1] H. Lashkari *et al.*, "Enhancing Computational Efficiency of Network Reliability with a New Prime Shortest Path Algorithm," vol. 13, p. 109, 2025, https://doi.org/10.3390/technologies13030109.

[2] Y. G. Narayana and V. Yegnanarayanan, "On Prime number varieties and their applications," *Engineering and Applied Science Letters*, vol. 3, no. 3, pp. 30–36, Dec. 2020, https://doi.org/10.30538/psrp-easl2020.0045.

[3] S. Sarna and R. Czerwinski, "Small prime divisors attack and countermeasure against the rsa-otp algorithm," *Electronics (Switzerland)*, vol. 11, no. 1, Jan. 2022, https://doi.org/10.3390/electronics11010095.

[4] J. A. Solinas, "Generalized Mersenne Prime," in *Encyclopedia of Cryptography, Security and Privacy*, Cham: Springer Nature Switzerland, 2025, pp. 1002–1004. https://doi.org/10.1007/978-3-030-71522-9_32.

[5] R. Bogotyrev, "Number theory and its applications in cybersecurity: a review," *Annals of Mathematics and Computer Science*, vol. 25, pp. 64–77, Nov. 2024, https://doi.org/10.56947/amcs.v25.370.

[6] B. Y. Wang and X. Wang, "Symmetrical Distribution of Primes and Their Gaps," *Advances in Pure Mathematics*, vol. 11, no. 05, pp. 447–456, 2021, https://doi.org/10.4236/apm.2021.115031.

[7] J. E. Cohen, "Gaps Between Consecutive Primes and the Exponential Distribution," *Exp Math*, vol. 34, no. 2, pp. 260–269, Apr. 2025, https://doi.org/10.1080/10586458.2024.2362348.

[8] P. Shiu, "Distribution of Prime Numbers," in *Number Theory with Computations*, Springer, 2024, pp. 235–269. https://doi.org/10.1007/978-3-031-63814-5_9.

[9] D. Lattanzi, "Statistical Distributions of Prime Number Gaps," *Journal of Advances in Mathematics and Computer Science*, vol. 39, no. 1, pp. 36–61, Jan. 2024, https://doi.org/10.9734/jamcs/2024/v39i11861.

[10] W. Banks, K. Ford, and T. Tao, "Large prime gaps and probabilistic models," *Invent Math*, vol. 233, no. 3, pp. 1471–1518, Sep. 2023, https://doi.org/10.1007/s00222-023-01199-0.

[11] G. Iovane, E. Benedetto, and C. Gallo, "Multiscale Sieve for Smart Prime Generation and Application in Info-Security, IoT and Blockchain," *Applied Sciences (Switzerland)*, vol. 14, no. 19, Oct. 2024, https://doi.org/10.3390/app14198983.

[12] A. Ezz-Eldien *et al.*, "Computational challenges and solutions: Prime number generation for enhanced data security," *PLoS One*, vol. 19, no. 11, p. e0311782, Nov. 2024, https://doi.org/10.1371/journal.pone.0311782.

[13] B. Tran and S. Vaudenay, "Solving the Learning Parity with Noise Problem Using Quantum Algorithms," in *International Conference on Cryptology in Africa*, Springer, 2022, pp. 295–322. https://doi.org/10.1007/978-3-031-17433-9_13.

[14] M. Wang, G. Huang, H. Gao, and L. Hu, "Improved Zero-Knowledge Proofs for Commitments from Learning Parity with Noise," in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, Dec. 2022, pp. 415–421. https://doi.org/10.1109/TrustCom56396.2022.00064.

[15] I. Papadakis, "Representation and Generation of Prime and Coprime Numbers by Using Structured Algebraic Sums," *Mathematics and Computer Science*, vol. 9, no. 3, pp. 57–63, Aug. 2024, https://doi.org/10.11648/j.mcs.20240903.12.

[16] M. Loconsole and L. Regolin, "Are prime numbers special? Insights from the life sciences," *Biol Direct*, vol. 17, no. 1, p. 11, Dec. 2022, https://doi.org/10.1186/s13062-022-00326-w.

[17] J. Sorenson and J. Webster, "An algorithm and computation to verify Legendre's conjecture up to $$7\cdot 10^{13}$$," *Res Number Theory*, vol. 11, no. 1, p. 4, Mar. 2025, https://doi.org/10.1007/s40993-024-00589-4.

[18] M. Ghidarcea and D. Popescu, "Classic Linear Sieves Revisited," *Journal of Control Engineering and Applied Informatics*, vol. 26, no. 4, pp. 3–14, Dec. 2024, https://doi.org/10.61416/ceai.v26i4.9098.

[19] H. M. Bahig, M. A. G. Hazber, K. Al-Utaibi, D. I. Nassr, and H. M. Bahig, "Efficient Sequential and Parallel Prime Sieve Algorithms," *Symmetry (Basel)*, vol. 14, no. 12, p. 2527, Nov. 2022, https://doi.org/10.3390/sym14122527.

[20] V. Umadevi and K. Sivakami, "Comparison and Performance Evaluation of Prime Number Generation Using Sieves Algorithm," *J Comput Theor Nanosci*, vol. 17, no. 4, pp. 1839–1841, Apr. 2020, https://doi.org/10.1166/jctn.2020.8450.

[21] A. M. Zaki, M. E. Bakr, A. M. Alsahangiti, S. K. Khosa, and K. A. Fathy, "Acceleration of Wheel Factoring Techniques," *Mathematics*, vol. 11, no. 5, p. 1203, Mar. 2023, https://doi.org/10.3390/math11051203.

[22] C. Fei, X. Zhang, D. Wang, H. Hu, R. Huang, and Z. Wang, "EPRNG: Effective Pseudo-Random Number Generator on the Internet of Vehicles Using Deep Convolution Generative Adversarial Network," *Information*, vol. 16, no. 1, p. 21, Jan. 2025, https://doi.org/10.3390/info16010021.

[23] S. H. AbdELHaleem, S. K. Abd-El-Hafiz, and A. G. Radwan, "Analysis and Guidelines for Different Designs of Pseudo Random Number Generators," *IEEE Access*, vol. 12, pp. 115697–115715, 2024, https://doi.org/10.1109/ACCESS.2024.3445277.

[24] M. De Bernardi, M. H. R. Khouzani, and P. Malacaria, "Pseudo-Random Number Generation Using Generative Adversarial Networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2019, pp. 191–200. https://doi.org/10.1007/978-3-030-13453-2_15.

[25] R. Oak, C. Rahalkar, and D. Gujar, "Poster," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: ACM, Nov. 2019, pp. 2597–2599. https://doi.org/10.1145/3319535.3363265.

[26] H. Kim, Y. Kwon, M. Sim, S. Lim, and H. Seo, "Generative Adversarial Networks-Based Pseudo-Random Number Generator for Embedded Processors," in *International Conference on Information Security and Cryptology*, Springer, 2021, pp. 215–234. https://doi.org/10.1007/978-3-030-68890-5_12.

[27] K. Okada, K. Endo, K. Yasuoka, and S. Kurabayashi, "Learned pseudo-random number generator: WGAN-GP for generating statistically robust random numbers," *PLoS One*, vol. 18, no. 6 June, Jun. 2023, https://doi.org/10.1371/journal.pone.0287025.

[28] X. Wu, Y. Han, S. Zhu, Y. Li, S. Cui, and X. Wang, "Learned Pseudo-Random Number Generator Based on Generative Adversarial Networks," in *International Conference on Frontiers in Cyber Security*, Springer, 2024, pp. 517–530. https://doi.org/10.1007/978-981-99-9331-4_34.

[29] X. Wu, Y. Han, M. Zhang, Y. Li, and S. Cui, "GAN-based pseudo random number generation optimized through genetic algorithms," *Complex & Intelligent Systems*, vol. 11, no. 1, p. 31, Jan. 2025, https://doi.org/10.1007/s40747-024-01606-w.

[30] A. Ünal, "Worst-Case Subexponential Attacks on PRGs of Constant Degree or Constant Locality," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2023, pp. 25–54. https://doi.org/10.1007/978-3-031-30545-0_2.

[31] A. Kumar and A. Mishra, "Evaluation of Cryptographically Secure Pseudo Random Number Generators for Post Quantum Era," in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, IEEE, Apr. 2022, pp. 1–5. https://doi.org/10.1109/I2CT54291.2022.9824543.

[32] R. Biswas, D. R. Talukdar, and U. Roy, "Exploring the Limits of Classical Random Number Generation and Unveiling Quantum Alternatives," in *International Conference on Network Security and Blockchain Technology*, Springer, 2025, pp. 35–45. https://doi.org/10.1007/978-981-97-8051-8_4.

[33] Y. Zhang, X. Lu, Y. Liu, Y. Yin, and K. Wang, "LEAP: High-Performance Lattice-Based Pseudorandom Number Generator," *IACR Transactions on Symmetric Cryptology*, vol. 2025, no. 3, pp. 151–182, Sep. 2025, https://doi.org/10.46586/tosc.v2025.i3.151-182.

[34] A. Saini, A. Tsokanos, and R. Kirner, "Quantum Randomness in Cryptography—A Survey of Cryptosystems, RNG-Based Ciphers, and QRNGs," *Information*, vol. 13, no. 8, p. 358, Jul. 2022, https://doi.org/10.3390/info13080358.

[35] C. Ryan, M. Kshirsagar, G. Vaidya, A. Cunningham, and R. Sivaraman, "Design of a cryptographically secure pseudo random number generator with grammatical evolution," *Sci Rep*, vol. 12, no. 1, Dec. 2022, https://doi.org/10.1038/s41598-022-11613-x.

[36] H. Halpin, "The adversary: the philosophy of cryptography," *J Cybersecur*, vol. 11, no. 1, 2025, https://doi.org/10.1093/cybsec/tyaf006.

[37] S. Senouci, S. A. Madoune, M. R. Senouci, A. Senouci, and Z. Tang, "A novel PRNG for fiber optic transmission," *Chaos Solitons Fractals*, vol. 192, p. 116038, Mar. 2025, https://doi.org/10.1016/j.chaos.2025.116038.