How Random is Random? A Comparative Machine Learning Evaluation on the Predictability of PRNG and CSPRNG Algorithms

Maheswara Bayu Kaindra – 13523015
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: kaindramaheswara11@gmail.com , 13523015@std.stei.itb.ac.id

***Abstract*—The security of most cryptographic systems hinges on the unpredictability of random number generators. While various Pseudo-Random Number Generators (PRNGs) and Cryptographically Secure PRNGs (CSPRNGs) are deployed across applications, the extent of their individual algorithmic predictability remains a critical security concern. This paper investigates "how random is random" by evaluating the predictability of several PRNG and CSPRNG algorithms using Machine Learning. By framing randomness as a sequence prediction task, this study utilizes Feed-Forward Neural Networks (FFNNs), 1-Dimensional Convolutional Neural Networks (1D CNNs), and Random Forest classifiers, implemented via Keras and PyTorch. These models are tasked with extracting underlying deterministic features and statistical patterns from the bitstreams generated by each specific algorithm. Experimental evaluations provide a granular comparative analysis, demonstrating the varying degrees to which individual algorithms either leak computationally learnable patterns or successfully resist predictive modeling. Ultimately, this study offers empirical insights into the resilience of specific generator algorithms against modern cryptanalysis.**

Keywords—PRNG; CSPRNG; Machine Learning; Randomness

I. Introduction

Random number generation is a fundamental component of modern cryptography, serving as the foundation for creating secure cryptographic keys, nonces, and initialization vectors [4]. In practical applications, systems heavily rely on Pseudo-Random Number Generators (PRNGs) due to their computational efficiency [2]. However, PRNGs are "inherently" deterministic; they produce sequences based on a mathematical formula and an initial seed value [4]. While standard PRNGs are sufficient for general computing tasks, this underlying determinism poses a critical vulnerability in security-sensitive contexts. If an attacker can predict future outputs based on observed sequences, the entire cryptographic system is compromised [3]. To mitigate this risk, Cryptographically Secure PRNGs (CSPRNGs) are specifically designed to withstand predictive modeling, ensuring that their generated sequences are computationally indistinguishable from true random noise [1], [5].

Determining the actual security and unpredictability of these generators requires rigorous evaluation. Traditionally, the randomness of an algorithm is assessed using standardized statistical test suites. However, the rapid advancement of Machine Learning (ML) introduces a modern and highly effective approach to cryptanalysis. Due to their pattern-recognition capabilities, ML algorithms can be utilized to detect subtle, non-linear deterministic features in seemingly random bitstreams that might pass conventional statistical evaluations.

II. Theoretical Background

A. Pseudo-Random Number Generator

Currently, no computational process has generated a truly random sequence of numbers. Computationally produced random numbers are inherently pseudo-random, meaning they are generated by deterministic mathematical algorithms that output exactly replicable sequences when initialized with the same seed. Such systems are known as pseudo-random number generators (PRNGs). Within modern cryptographic architectures, PRNGs serve as a critical structural component. They supply the necessary computational entropy to construct Initialization Vectors (IVs) in block cipher algorithms.

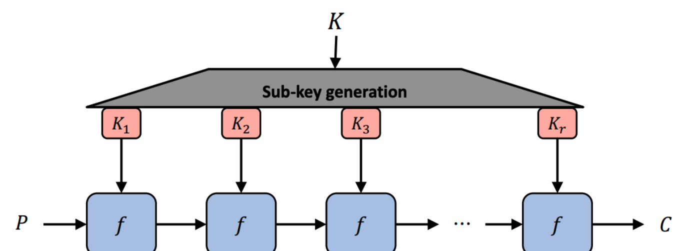

Fig. 1. AES as a block cipher algorithm utilizing PRNGs.
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/34-Pembangkit-bilangan-acak-2026.pdf (Munir, 2026)

Despite their operational necessity, the underlying deterministic nature of PRNGs creates a major security problem. If a cryptanalyst successfully infers the internal state, extracts the initial seed, or maps the pattern of the generator, the sequence becomes computationally predictable. Such predictability allows the attacker to anticipate future bits, compromising the confidentiality and integrity of the relying cryptographic system.

- *Confidentiality*: The assurance that sensitive data is accessible exclusively to authorized parties.
- *Integrity*: The assurance that data is not subjected to unauthorized alteration, manipulation, modification, or corruption during transmission or storage. The received message must be guaranteed perfectly identical to the one sent by the original source.

B. Linear Congruential Generator (LCG)

The Linear Congruential Generator is a pseudo-random number generator defined by the following equation.

$$x_{n+1} = (ax_n + b) \bmod m$$

- x_{n+1} = the current random number
- x_n = the previous random number
- a = the multiplier ($0 < a < m$)
- m = the modulus ($m > 0$)
- b = the increment ($0 < b < m$)
- x_0 = the secret seed

Consequently, the generated x_i values are bounded between 0 and $m - 1$. LCGs generally exhibit a small period, often strictly less than m. An LCG achieves a full period $(m - 1)$ if it satisfies the following conditions:

1. b is relatively prime to m
2. $a - 1$ is divisible by all prime factors of m
3. $a - 1$ is a multiple of 4 if m is a multiple of 4
4. $m > max(a, b, x_0)$
5. $a > 0$ and $b > 0$

C. XorShift32

Xorshift32 is a class of random number generators (RNGs) designed to be extremely fast and simple. The algorithm operates by repeatedly performing an exclusive-or (XOR) on a computer word with a shifted version of itself. Xorshift32 produces a sequence of $2^{32} - 1$ integers. The basic operations used are:

1. $y \wedge (y << a)$ (left shift)
2. $y \wedge (y >> a)$ (right shift).

Mathematically, Xorshift models the seed as a set of $1 \times n$ binary vectors (for Xorshift32, $n = 32$), excluding the zero vector. The elements of these vectors are in the field $(0, 1)$, meaning the addition of binary vectors can be implemented by XORing the constituent 32-bit parts. The generation relies on a linear transformation over the binary vector space, characterized by a nonsingular $n \times n$ binary matrix T. For a nonsingular binary matrix T to produce all possible non-full binary vectors in sequence, the matrix T must have an order of $2^n - 1$ in the group of nonsingular $n \times n$ binary matrices. Bit-shift operations can be represented as binary matrices:

- Let L be the $n \times n$ binary matrix that effects a left shift of one position (containing 1s on the principal subdiagonal and 0s elsewhere). The XOR and left shift operation is equivalent to the matrix transformation:

$$T = I + L^a$$

- Similarly, let R be the right shift matrix (the transpose of L). The XOR and right shift operation is equivalent to the matrix operation:

$$T = 1 + R^b$$

$$L = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

Fig. 2. L Matrix Illustration

The matrices $T = I + L^a$ and $T = 1 + R^b$ are nonsingular. However, for 32-bit architectures, a combination of only two shifts, such as $T = (I + L^a)(1 + R^b)$, does not provide a matrix with the required order to achieve the maximum period. Therefore, Xorshift32 uses three consecutive shift operations to form a matrix T that has an order of $2^{32} - 1$:

$$T = (I + L^a)(1 + R^b)(1 + L^c)$$

There are exactly 81 combinations of the triplet (a, b, c) with $a < c$ that produce a full period of this 32×32 binary matrix. By considering transpositions and different operational sequences (for example, right-left-right instead of left-right-left), there is a total of 648 valid parameter choices to create a Xorshift32 with a period of $2^{32} - 1$.

D. Cryptographically Secure PRNGs

To address the vulnerabilities of basic PRNGs, cryptography relies on Cryptographically Secure Pseudo-Random Number Generators (CSPRNG). Although CSPRNGs remain fundamentally deterministic pseudo-random generators, they must fulfill strict security

parameters, specifically passing statistical randomness tests and withstanding serious attacks aimed at predicting future generated values from previous outputs.

E. ChaCha20

ChaCha20 is a 256-bit stream cipher and a variant of the 20-round Salsa20/20 cipher. Designed by Daniel J. Bernstein, the cipher operates by expanding a 256-bit key into 2^{64} randomly accessible streams, where each stream comprises 2^{64} randomly accessible 64-byte blocks. The objective behind the ChaCha modifications is to maximize structural diffusion per round. This design increases resistance to cryptanalysis while preserving, and frequently improving, computational execution speed compared to its predecessors.

The building block of ChaCha20 is the quarter-round function. It invertibly updates four 32-bit state words (a, b, c, d) using addition modulo 2^{32}, bitwise XOR (⊕), and constant-distance bitwise rotation (<<<). ChaCha20 updates each state word twice per quarter-round. The rotation distances are 16, 12, 8, and 7 bits. The execution is defined as:

- $a = a + b;\ d = d \oplus a;\ d = d <<< 16$
- $c = c + d;\ b = b \oplus c;\ b = b <<< 12$
- $a = a + b;\ d = d \oplus a;\ d = d <<< 8$
- $c = c + d;\ b = b \oplus c;\ b = b <<< 7$

ChaCha20 organizes its initial state into a 4 × 4 grid, containing sixteen 32-bit words. The layout is specifically designed to place the attacker-controlled input words safely at the bottom of the grid. The matrix is filled row by row:

TABLE. 1. CHACHA20 INITIAL STATE

Rows	**Columns**			
	0	***1***	***2***	***3***
0	Constant	Constant	Constant	Constant
1	Key	Key	Key	Key
2	Key	Key	Key	Key
3	Input	Input	Input	Input

F. A5/1

The A5/1 cipher is a synchronous stream cipher standard primarily utilized for encrypting voice transmission signals within the Global System for Mobile Communications (GSM) standard. As the stronger variant of the A5 algorithm family, A5/1 was originally kept secret but was publicly reverse-engineered in 1994. It generates a 228-bit keystream designed to be XORed with a 228-bit communication frame transmitted every 4.6 milliseconds.

The internal state of A5/1 comprises three distinct Linear Feedback Shift Registers (LFSRs) with lengths of 19, 22, and 23 bits, yielding a total internal state of 64 bits. This 64-bit state corresponds to the length of the external session key used in the protocol. The final pseudo-random output is derived from the bitwise XOR operation of the most significant bits shifted out from these three registers.

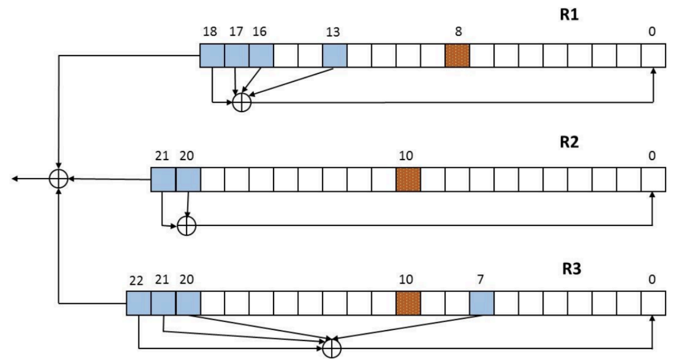

Fig. 3. Linear Feedback Shift Registers Used in A5/1
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/13-Stream-Cipher-STI-2026.pdf (Munir, 2026)

To clock each register, the following conditions apply:

1. The 8th bit of Register 1, alongside clocking bits located at positions 13, 16, 17, and 18.

   $$new\ bit = bit[13] \oplus bit[16] \oplus bit[17] \oplus bit[18]$$

2. The 10th bit of Register 2, alongside clocking bits located at positions 20 and 21.

   $$new\ bit = bit[20] \oplus bit[21]$$

3. The 10th bit of Register 3, alongside clocking bits located at positions 7, 20, 21, and 22.

   $$new\ bit = bit[7] \oplus bit[20] \oplus bit[21] \oplus bit[22]$$

G. System SecureRandom

Per official Java specifications, a cryptographically strong RNG must minimally comply with the statistical randomness test delineated in FIPS 140-2, Security Requirements for Cryptographic Modules. The java.security.SecureRandom class provides a Cryptographically Strong Pseudo Random Number Generator (CSPRNG). Unlike standard generators, SecureRandom is strictly mandated to produce non-deterministic output, ensuring that all output sequences and seed materials satisfy the rigorous unpredictability constraints described in RFC 4086. The cryptographic resilience of a CSPRNG is contingent upon its initial entropy.

1. Self-Seeding: A newly instantiated SecureRandom object (excluding those constructed with an explicit byte array) remains unseeded. The primary invocation of the nextBytes() method forces the object to automatically extract initial entropy from an

OS-level, implementation-specific source (e.g., /dev/urandom or /dev/random in Unix-like systems).

2. State Reseeding: To mitigate internal state compromise over extended lifecycles, the internal state can be periodically refreshed. The setSeed() method permits the manual injection of external, unpredictable seed material. Furthermore, the reseed() method extracts fresh input directly from the implementation's underlying entropy source to autonomously update its state.

SecureRandom operates within the Java Cryptography Architecture (JCA) framework via a Service Provider Interface (SecureRandomSpi). It employs a factory pattern (getInstance()) that traverses registered security Provider objects to resolve specific algorithmic implementations. Modern Java implementations allow service providers to explicitly advertise thread-safe capabilities (via the "ThreadSafe" attribute), permitting concurrent extraction of random bytes in multi-threaded environments without degrading the statistical distribution or predictability of the output sequence.

H. Feedforward Neural Network (FFNN)

The FFNN is one of foundational deep learning architecture where information propagates in a single, unidirectional flow from the input layer, through hidden layers, to the output layer. In next-bit prediction, the FFNN acts as an algebraic solver tasked with learning the generator's state-transition function.

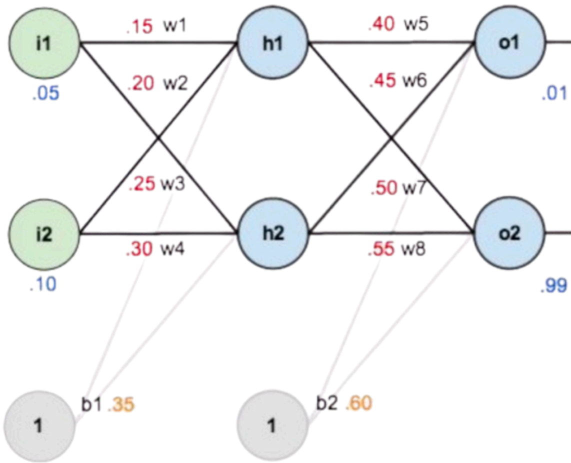

Fig. 4 FFNN Architecture Example
Source: ITB ML Lecture Slides (IF3270 Lecturer Team, 2026)

Given a sliding window input vector of w bits, $X = [b1, b2, ..., bw]$, each artificial neuron computes a weighted sum of its inputs and applies a non-linear activation function (such as ReLU or Sigmoid):

$$y = f(\sum_{i=1}^{w} w_i x_i + \beta)$$

The hidden layers allow the FFNN to map complex mathematical relationships. FFNN learns by iteratively adjusting the weights via backpropagation.

I. Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are deep learning architectures designed to process structured topological data, such as one-dimensional sequential arrays. The core mechanism of a CNN relies on the mathematical operation of convolution, which replaces dense, global matrix multiplications with shift-invariant feature extraction.

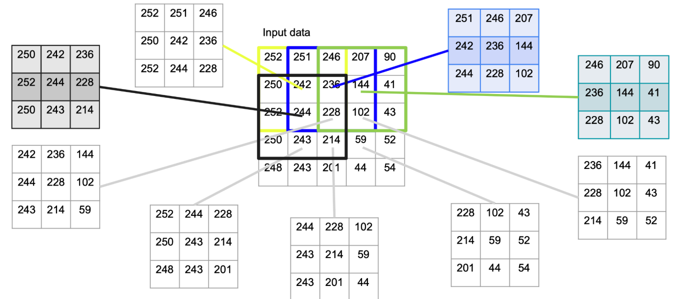

Fig. 5. Receiptive Field for Grid-like Data with Kernel 3*3
Source: ITB ML Lecture Slides (IF3270 Lecturer Team, 2026)

In the context of pseudo-randomness evaluation, a 1D CNN is utilized to test the hypothesis that structural dependencies within a bitstream can be captured as localized spatial patterns. The network deploys a series of learnable filters (kernels) of size k that slide across the sequential input. At each step, a dot product is computed between the filter weights and the localized data segment:

$$C_j = f(\sum_{m=0}^{k-1} w_m x_{j+m} + \beta)$$

Theoretically, if a generator's underlying mathematical structure (such as the LFSRs in the stream ciphers) produces recurring frequency biases, the filters can isolate these mathematical signatures regardless of their positional shift in the sequence. Following this extraction, pooling layers down-sample the spatial dimensionality to condense the most prominent features. Finally, a fully connected dense layer synthesizes this high-level representation to perform the classification task.

III. Implementation Design

A. System Architecture Pipeline

The evaluation framework is constructed as a decoupled, multi-language (Java-Python) data pipeline. The data flow is unidirectional across three main phases, separated by file Input/Output (I/O) Boundaries.

1. Data Generation Phase (Java)

   The pipeline originates in a Java-based environment (providing access to java.security.secureRandom class. The Java module operates as an independent random number sequence generator.

2. Preprocessing Phase (Python)

Operating within Python, this phase reads the raw outputs (.txt) generated by the Java module and transforms them into high-dimensional binary tensors. This Phase is responsible for feature engineering, mapping the continuous stream into supervised learning problem using a sliding window mechanism.

3. Model Training and Evaluation Phase (Python Machine Learning Environment)

The evaluation phase ingests the pre-computed binary tensors from the second phase. This environment focuses on training the predictive models (Random Forest, FFNN, and 1D CNN) and extracting standard classification metrics (Accuracy and ROC-AUC).

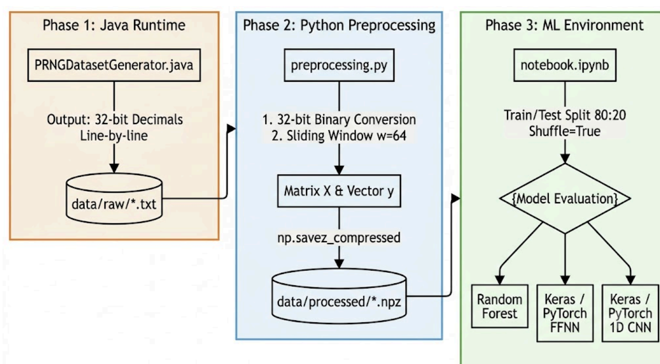

Fig. 6. System Architecture Pipeline
Made with DrawIO

To align with this architecture, the repository is divided into two main directories: ml-evaluator and prng-generator. The prng-generator directory houses a core file, PRNGDatasetGenerator.java, which implements the Random Number Generator classes (LCG, Xorshift32, A5/1, ChaCha20, and SecureRandom). Meanwhile, ml-evaluator contains all files related to the Machine Learning environment, spanning from data preprocessing to modeling.

B. Data Generation

The data generation process is handled by PRNGDatasetGenerator.java. Following Java's structural paradigms, the PRNGDatasetGenerator class contains a PRNG interface, which is implemented by underlying static classes for the selected algorithms: LCG, Xorshift32, A5/1, ChaCha20, and SecureRandom (Java native).

For each algorithm, the subsequent sequence prediction is handled by the overridden method from the PRNG interface, with details as follows.

1. LCG

```
FUNCTION next:
    1.  Multiply the current state by a fixed
        multiplier (a).
    2.  Add a fixed increment (c) to the result.
    3.  Divide the sum by a fixed maximum limit
        (m) and keep only the remainder.
    4.  Save this remainder as the new state.
    5.  RETURN the new state.
```

2. Xorshift32

```
FUNCTION next:
    1.  Take the current state, shift its bits to
        the LEFT by 13, and XOR it with the
        original state.
    2.  Take the new state, shift its bits to the
        RIGHT by 17, and XOR it with itself.
    3.  Take the newest state, shift its bits to
        the LEFT by 5, and XOR it with itself.
    4.  Ensure the final result stays within a
        32-bit limit.
    5.  RETURN the final state.
```

3. ChaCha20

```
FUNCTION next_ChaCha20:
    1.  IF all 16 numbers in the current 'buffer'
        have been used:
            a.  CALL generate_new_block(): Copy the
                core 'state' into a temporary
                'buffer'.
            b.  Perform 20 rounds of mixing (using
                addition, XOR, and rotation) on the
                'buffer'.
            c.  Add the original core 'state'
                values to the mixed 'buffer'.
            d.  Increase the block counter so the
                next block will be different.
            e.  Reset the usage index to 0.
    2.  Get the next available number from the
        'buffer' using the current index.
    3.  Increase the index by 1.
    4.  RETURN the extracted number.
```

4. A5/1

```
FUNCTION next:
    1.  Set the 'final_output' variable to 0.
    2.  REPEAT 32 times:
            a.  Look at clocking bits in r1, r2,
                and r3.
            b.  Determine the majority value (0 or
                1) among these three bits.
            c.  IF register r1's clocking bit
                equals the majority, Shift r1 and
                update it using XOR on specific
                internal bits.
            d.  IF register r2's clocking bit
                equals the majority, Shift r2 and
                update it using XOR on specific
                internal bits.
            e.  IF register r3's clocking bit
                equals the majority:
```

```
            Shift r3 and update it using XOR on
            specific internal bits.
        f.  Create a single 'output_bit' by
            XOR-ing the top bits of r1, r2, and
            r3.
        g.  Append 'output_bit' to the
            'final_output'.
    3.  RETURN the 'final_output'.
```

4. SecureRandom

```
FUNCTION next_SecureRandom:
    1.  Ask the system's built-in secure random
        generator to provide a random integer.
    2.  Convert it to ensure it behaves as a
        positive 32-bit unsigned number.
    3.  RETURN the number.
```

The class hierarchy is depicted in the following figure.

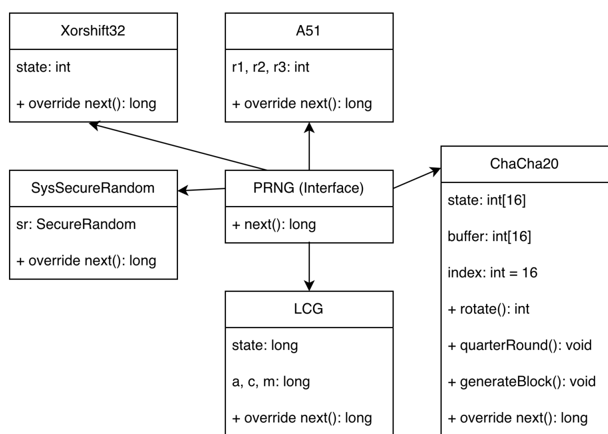

Fig. 7. PRNGDatasetGenerator Class Diagram
Made with DrawIO

C. Data Preprocessing (preprocessing.py)

The preprocessing module is responsible for transforming the raw decimal logs into structured, high-dimensional binary tensors required by the machine learning architectures. This transformation frames the randomness evaluation as a supervised next-bit prediction task.

1. Binary Sequence Reconstruction

   The initial preprocessing step handles the conversion of numerical sequences back into their pure bitstream representation. The script reads the decimal sequences from the .txt files line-by-line. Each decimal integer is formatted into a 32-bit binary string (using Python's 032b format specifier) to preserve leading zeros, ensuring exact structural parity with the original 32-bit register outputs of the Java generators. These strings are concatenated into a single, continuous full-bit string and subsequently cast into a 1-Dimensional NumPy array of 8-bit integers (np.int8) to optimize matrix operations.

2. Sliding Window Transformation and Labeling

   To train the supervised learning models, the continuous 1D bit array must be mapped into overlapping discrete features and target lables. The system utilizes NumPy's sliding_window_view() function to extract structural dependencies from the stream efficiently.

   A fixed window size of $w = 64$ bits is established, determining the length of the historical sequence the model observes to make a prediction. The transformation operates as follows:

   a. The sliding window extracts arrays of $w + 1$ bits (65 bits) per step, shifting by a stride of 1 bit.
   b. The extracted arrays are sliced to separate the predictors from the target. The first 64 bits (windows[:, :-1]) are assigned to the feature matrix X, representing the internal state sequence of the PRNG.
   c. The 65th bit (windows[:, -1]) is isolated as the target vector y, representing the bit the ML model must predict (ML model will predict the last bit).

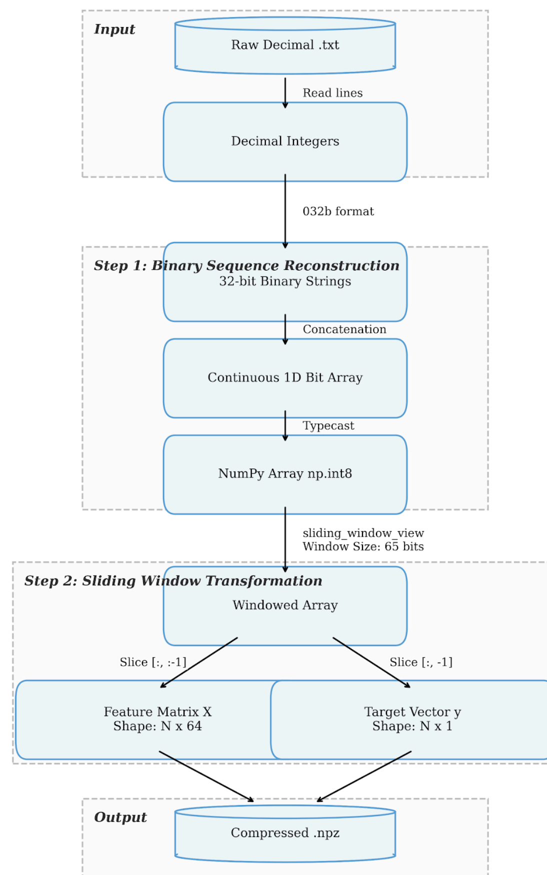

Fig. 8. Data Preprocessing Pipeline
Python Generated

The resulting matrix X and vector y are serialized and compressed into an .npz archive (np.saves_compressed). This compression (should) reduce disk storage requirements

significantly and accelerates memory loading times during the model training phase.

D. Modeling & Evaluation Design

The predictive modeling phase ingest the preprocessed feature matrix X and target labels y to evaluate the cryptographic sequences. The models are tasked with identifying non-linear deterministic patterns within the 64-bit windows. The evaluation utilizes a baseline ensemble method alongside deep learning architectures implemented in both Keras and PyTorch.

1. Random Forest

   The model is configured with 100 estimators (n_estimators=100) and a constrained maximum depth (max_depth=10).

2. Feed-Forward Neural Network (FFNN)

   The FFNN is implemented across Keras and PyTorch with the following topology.

   a. Input Layer: Accepts the flat 64-bit feature vector.
   b. Hidden Layers: Two dense layers consisting of 128 and 64 neurons, respectively, utilizing Rectified Linear Unit (ReLU) activation functions to capture non-linear relationships.
   c. Output Layer: A single neuron utilizing a Sigmoid activation function to output the binary classification probability of the subsequent bit.

3. 1-Dimensional Convolutional Neural Network (1D CNN)

   The input vectors are reshaped to represent a single-channel sequence. The architecture comprises:

   a. Convolutional Block: A 1D Convolutional layer featuring 32 filters and a kernel size of 3, paired with a ReLU activation function. This is followed by a 1D Max Pooling layer with a pool size of 2 to down-sample the spatial dimensionality.
   b. Fully Connected Block: The extracted feature maps are flattened and passed through a 64-neuron dense layer (ReLU), concluding with a single-neuron Sigmoid output layer.

Both deep learning architectures are compiled using the Adam optimizer with a learning rate of 0.001 and Binary Cross-Entropy (BCE) as the loss function. Training is executed over 5 epochs using a batch size of 256.

1. Data Splitting and Randomization

   Prior to model ingestion, the windowed datasets are partitioned into an 80% training set and a 20% testing set (test_size=0.2).

2. Predictability Metrics

   The extent to which an algorithm leaks computationally learnable patterns is measured using two primary metrics on the test set:

   a. Accuracy Score

      Evaluates the absolute proportion of correct next-bit predictions made by the models. The continuous probability outputs from the Sigmoid layer are binarized using 0.5 threshold.

   b. ROC-AUC

      Assesses the model's aggregate capability to distinguish between the probability distributions of the '0' and '1' classes, independent of the binarization threshold.

A perfectly secure and unpredictable sequence must yield prediction metrics asymptotically close to 0.5 (50%). Any significant statistical deviation above this threshold demonstrates that the algorithm exhibits predictive vulnerabilities under ML analysis.

The complete source code is available in the following GitHub repository. Across all configurations, each generator was initialized with a fixed seed of 12345 to produce a standardized dataset of 10,000 sequential bits.

IV. Results and Analysis

This experiment evaluates the predictive performance of five Machine Learning architectures (Random Forest, Keras FFNN, Keras 1D CNN, PyTorch FFNN, and PyTorch 1D CNN) in forecasting the next bit of pseudo-random sequences generated by five distinct algorithms. Performance is measured using Accuracy and Area Under the ROC Curve (ROC-AUC). In a cryptanalytic context, the baseline for a random guess is 0.5000 (50%); therefore, scores consistently exceeding this threshold indicate deterministic pattern leakage from the underlying generator. The prediction results of the models are presented in the following tables:

Table. 2. Model Prediction Accuracy (%)

Model	PRNGs				
	LCG	***Xorshift***	***A5/1***	***ChaCha***	***Secure Random***
Random Forest	0.5516	0.5546	0.5014	0.5045	0.5000
Keras FFNN	0.5550	0.6973	0.4996	0.5040	0.5021
Keras 1D CNN	0.5498	0.5037	0.4979	0.5010	0.5025
PyTorch FFNN	0.5617	0.7043	0.5002	0.4980	0.5008
PyTorch 1D CNN	0.5485	0.4964	0.4987	0.5000	0.5024

Table. 3. Model Prediction ROC-AUC

Model	PRNGs				
	LCG	***Xorshift***	***A5/1***	***ChaCha***	***Secure Random***
Random Forest	0.5751	0.5772	0.5027	0.5062	0.5027

Model	PRNGs				
	LCG	***Xorshift***	***A5/1***	***ChaCha***	***Secure Random***
Keras FFNN	0.5853	0.7728	0.5041	0.5037	0.5011
Keras 1D CNN	0.5696	0.5048	0.4977	0.5044	0.5028
PyTorch FFNN	0.5898	0.7780	0.5015	0.5021	0.5006
PyTorch 1D CNN	0.5642	0.5005	0.5001	0.5000	0.5030

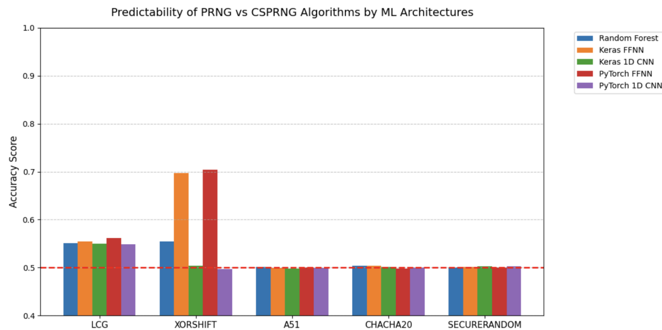

Fig. 9. Predictability of PRNG vs CSPRNG Algorithms by ML Architectures Python Generated

Based on the data presented in Tables 2 and 3, the generator algorithms can be classified into three levels of computational resilience: Vulnerable (LCG, Xorshift32), Small-Scale Resistant (A5/1), and Fully Cryptographically Resistant (ChaCha20, SecureRandom).

Experimental results demonstrate that the LCG fails the Next-Bit Test. All model architectures consistently achieved accuracies ranging from 54.8% to 56.1%, which constitutes massive information leakage (deviating significantly from the 50% baseline). The LCG operates on a simple linear algebraic equation, $x_{n+1} = (ax_n + b) \, mod \, m$. By only relying on multiplication and addition without complex non-linear transformations, feature mappings across sequential windows become highly susceptible to extraction. Consequently, even ensemble architectures such as Random Forest (Accuracy: 0.5516) can construct sufficiently accurate decision boundaries to partition this algebraic space.

The most significant finding of this experiment belongs to the Xorshift32 algorithm. The PyTorch FFNN model achieved a record high accuracy of 70.43% (ROC-AUC: 0.7780), indicating that the algorithm is highly predictable. Somehow, the 1D CNN architectures (both Keras and PyTorch) failed completely, yielding scores between 49.6% and 50.3%, which is potentially random guessing. Xorshift32 utilizes linear operations over Galois fields, specifically bit-shifting and XOR. By employing Dense layers, the FFNN observes the entire 64-bit window globally and simultaneously. This architecture allows the neural network to rapidly model the linear transformation matrices inherent in the bit-shift operations, enabling it to 'learn' the Xorshift formula almost perfectly. Besides of that, CNN architectures utilize local spatial filters (e.g., kernel size = 3), excelling at pattern recognition when correlated bits are in close proximity. However, because Xorshift32's shift operations displace bit values across multiple indices simultaneously (e.g., <<13 and >>17), the local spatial correlation might be disrupted. This explains why CNNs remain blind to Xorshift32 patterns.

Although A5/1 is a legacy stream cipher deemed compromised by modern cryptographic standards, this experiment demonstrates that its architecture remains significantly more secure than purely non-cryptographic PRNGs (LCG and Xorshift32). All computational models plateaued at an accuracy of approximately 50.1%, failing to extract any meaningful predictive patterns. Unlike LCG or Xorshift32, which directly expose their linear algebraic mappings, A5/1 distributes its internal state across three Linear Feedback Shift Registers (LFSRs). The intervention of a majority clocking function to control these register shifts introduces a dynamic non-linear layer to every output cycle.

ChaCha20 and SecureRandom are proven to be immune to temporal predictive analysis. All models strictly stagnated at around 50.0%, statistically equivalent to a coin toss. The ARX (Add-Rotate-XOR) operations and quarter-round structures introduce extreme confusion and diffusion properties. A single-bit alteration in the input changes an average of 50% of the output bits. Consequently, neither the dense layers of the FFNN nor the convolutional filters of the CNN can discover any mathematical gradients linking sequential windows. Unlike the other deterministic algorithms that rely on an initial algebraic seed, SecureRandom continuously harvests pure entropy from OS hardware noise. Due to its genuinely physical and stochastic foundation, there is no hidden mathematical function for Machine Learning models to reconstruct, establishing it as the absolute baseline for true randomness security in this study.

V. Conclusion

This study evaluated the vulnerability of random number generators to Machine Learning using Next-Bit Prediction. Results show that predictability is inversely proportional to an algorithm's non-linearity and entropy. Basic PRNGs like the Linear Congruential Generator (LCG) are highly susceptible, with all tested models successfully mapping its linear decision boundaries and exceeding the 50% random-guess baseline.

A key novelty is the architectural anomaly observed when evaluating Xorshift32, demonstrating that successful AI cryptanalysis depends heavily on the predictor model's structural fit. Feedforward Neural Networks (FFNNs) successfully compromised Xorshift32 (achieving 70.43% accuracy) because their dense layers capture global linear transformations in Galois space. Conversely, 1D Convolutional Neural Networks (CNNs) failed completely; large spatial bit-shifts destroy the local correlations required by convolutional filters. This confirms that no single deep learning architecture is universally effective for cryptanalysis.

Cryptographically, the LFSR-based A5/1 cipher exhibited moderate resistance, as its majority clocking mechanism disrupted feature extraction over limited samples. Furthermore, the security of CSPRNGs (ChaCha20 and System SecureRandom) was empirically validated. ChaCha20's extreme ARX diffusion and SecureRandom's

hardware entropy harvesting degraded all predictive models to pure random guessing (~50%). This confirms that non-deterministic systems lacking closed mathematical correlations remain immune to spatial and temporal feature extraction.

Acknowledgment

A. English

The author expresses sincere gratitude to everyone who provided support during the II4021 Cryptography course for the 2025/2026 academic year—including the lecturer, colleagues, and peers—as well as those who offered valuable feedback throughout the drafting of this paper. Recognizing that this study remains open to improvement, the associated research repository has been made publicly accessible to encourage further exploration and refinement by the broader community.

B. Indonesian

Penulis mengucapkan terima kasih kepada pihak-pihak yang telah membantu penulis selama melaksanakan perkuliahan II4021 Kriptografi pada tahun ajaran 2025/2026 (Dosen, rekan kerja, hingga teman belajar) dan mereka yang telah memberikan saran selama proses penulisan makalah ini. Penulis menyadari bahwa makalah ini masih jauh dari sempurna. Oleh karena itu, repositori dari penelitian ini bersifat terbuka bagi siapa saja yang ingin melakukan eksplorasi atau penyempurnaan lebih lanjut.

References

[1] Bernstein, D. J. (2008). *ChaCha, a variant of Salsa20*. The University of Illinois at Chicago

[2] Marsaglia, G. (n.d.). *Xorshift RNGs*. The Florida State University

[3] Munir, R. (2026a). *13 - Stream cipher*. Program Studi Sistem dan Teknologi Informasi, Institut Teknologi Bandung

[4] Munir, R. (2026b). *Pembangkit bilangan acak*. Program Studi Sistem dan Teknologi Informasi, Institut Teknologi Bandung

[5] Oracle. (n.d.). *Class SecureRandom*. Java™ Platform, Standard Edition 8 API Specification. Accessed from https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

Appendix

The complete source code is available in the following GitHub repository.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19th of June, 2026

Maheswara Bayu Kaindra (13523015)