Rapport de Projet : Évaluation de la Fiabilité Cryptographique des PRNG

Illan MAKHLOUFI

Décembre 2026

Table des matières

1 Introduction et Contexte

La génération de nombres aléatoires est une pierre angulaire de la sécurité informatique moderne. Si l'intuition humaine perçoit le hasard comme une simple absence de motif, en informatique, ce concept se divise en deux réalités distinctes : l'aléatoire statistique (utile pour la simulation) et l'aléatoire cryptographique (nécessaire pour la sécurité).

Les nombres aléatoires sont utilisés pour générer des clés de chiffrement, des vecteurs d'initialisation (IV), des nonces ou des jetons de session. Une faiblesse dans le générateur permettrait à un attaquant de prédire ces valeurs, compromettant l'intégralité du système, indépendamment de la robustesse de l'algorithme de chiffrement utilisé (comme AES ou RSA).

Ce projet a pour objectif d'analyser, d'implémenter et d'attaquer différentes familles de générateurs (PRNG et CSPRNG) afin de démontrer que *"passer les tests statistiques n'est pas une preuve de sécurité."*

2 Étude et Implémentation des Générateurs

Nous avons implémenté cinq algorithmes représentatifs des différentes approches de la génération aléatoire.

2.1 Linear Congruential Generator (LCG)

Le LCG est l'un des plus anciens algorithmes (1951). Il repose sur une récurrence linéaire simple :

$$X_{n+1} = (a \cdot X_n + c) \mod m \tag{1}$$

Analyse : Bien qu'extrêmement rapide, le LCG souffre de graves défauts structurels. Dans un espace à plusieurs dimensions, les points générés ne remplissent pas l'espace uniformément mais s'alignent sur des hyperplans (effet de réseau). Il est trivialement prédictible.

2.2 Mersenne Twister (MT19937)

C'est le générateur par défaut de nombreux environnements (Python `random`, Excel, PHP). Il est basé sur une récurrence matricielle dans un corps fini binaire ($\mathbb{F}_2$). **Analyse :** Il offre une période gigantesque ($2^{19937} - 1$) et passe la plupart des tests statistiques d'uniformité. Cependant, il n'est pas conçu pour résister à une analyse cryptographique : observer son état permet de prédire toutes les sorties futures.

2.3 Blum Blum Shub (BBS)

Le BBS est un exemple académique de CSPRNG (Cryptographically Secure PRNG).

$$X_{n+1} = X_n^2 \mod M \quad \text{où } M = p \cdot q \tag{2}$$

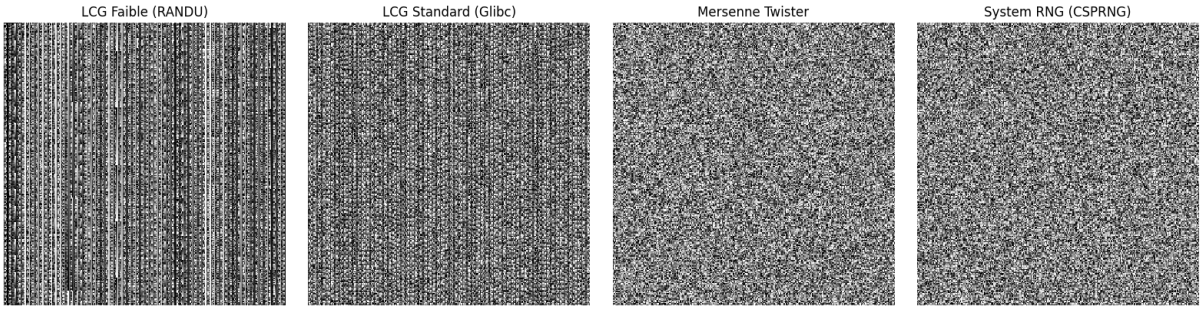
Analyse : Sa sécurité repose sur la difficulté du problème de la résiduosité quadratique, lié à la factorisation de grands nombres entiers. Bien que très lent, il offre une preuve de sécurité mathématique forte.

2.4 SystemRNG (Wrapper os.urandom)

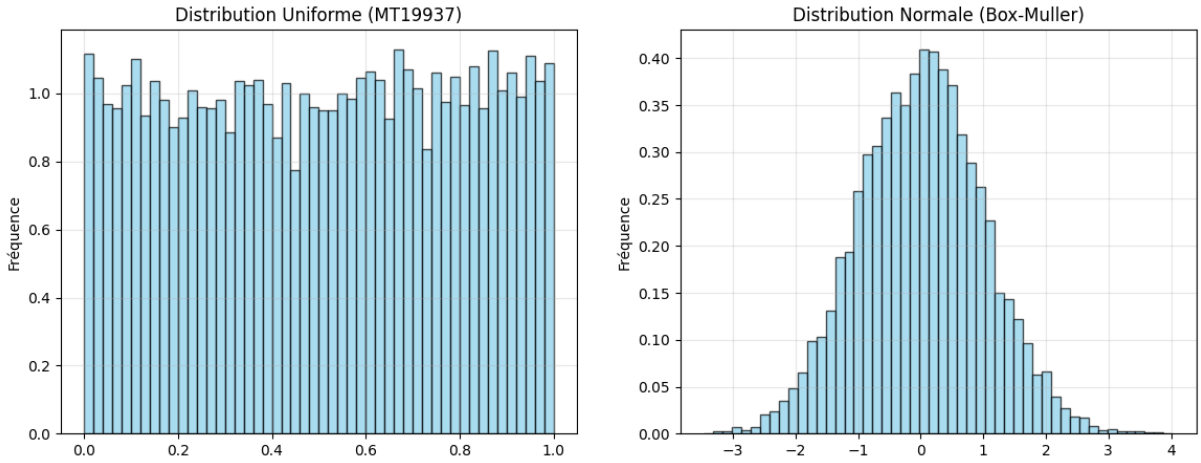
Ce générateur n'utilise pas d'algorithme déterministe pur, mais puise dans le pool d'entropie du système d'exploitation (interruptions matérielles, mouvements de la souris, bruit thermique des composants). C'est la référence absolue pour la cryptographie appliquée.

3 Analyse Visuelle (Bitmap Test)

Une méthode intuitive pour évaluer la qualité d'un générateur est de transformer ses sorties en image. Chaque octet généré définit le niveau de gris d'un pixel. Une superposition verticale permet ici de bien distinguer les détails des artefacts.



(a) **LCG (Mauvais paramètres)** : Apparition nette de motifs diagonaux et de répétitions. Ces lignes montrent que les valeurs sont fortement corrélées.



(b) **SystemRNG / MT19937** : Bruit blanc caractéristique ("neige télévisuelle"). Aucune structure n'est visible à l'œil nu, indiquant une bonne uniformité statistique.

FIGURE 1 – Comparaison structurelle de l'aléatoire.

L'analyse visuelle (Fig. 1) permet de disqualifier immédiatement le LCG. Les motifs visibles indiquent que connaissant un pixel, on peut deviner ses voisins avec une probabilité supérieure au hasard.

4 Résultats des Tests Statistiques

Nous avons soumis six configurations de générateurs à notre suite de tests. Le tableau ci-dessous synthétise les métriques clés : le score global (moyenne pondérée), l'entropie de Shannon et les résultats des tests d'hypothèse (Chi-2 et Kolmogorov-Smirnov).

Générateur	**Score Global** (max 1.0)	**Entropie** (bits/octet)	**Chi-2 (P-val)** (H_0 : Uniforme)	**Verdict χ^2**	**KS (P-val)**
LCG (Standard)	0.738	7.9522	0.0000	**Échec**	0.0000
LCG (Faible/RANDU)	0.695	7.7091	0.0000	**Échec**	0.0000
Mersenne Twister	**0.999**	7.9958	0.0432	Succès	0.0291
XOR Hybride	0.996	7.9961	0.2416	Succès	0.0005
System RNG (Crypto)	**0.997**	**7.9970**	0.9834	Succès	0.0118
Blum-Blum-Shub (BBS)	0.988	7.9611	0.2409	Succès	0.3728

TABLE 1 – Synthèse des analyses statistiques (Données issues du notebook de benchmark)

4.1 Interprétation des Métriques

Pour bien comprendre ces résultats, il faut définir ce que nous mesurons :

- **Entropie :** Elle mesure la quantité d'information imprévisible. L'idéal est de 8 bits par octet.
- **P-value (Chi-2 & KS) :** C'est la probabilité que les données soient uniformes par pur hasard.
  - Si $p < 0.01$ (1%), on rejette l'hypothèse nulle : le générateur est **biaisé** (Échec).
  - Si $p > 0.01$, le générateur est considéré comme statistiquement uniforme.

4.2 Analyse des Résultats

4.2.1 L'échec des générateurs linéaires (LCG)

Les deux versions du LCG (Standard et Faible) affichent des P-values de 0.0000 pour le test du Chi-2 et le test KS.

Signification : La distribution des octets n'est absolument pas uniforme. Il y a des "trous" et des répétitions dans les valeurs produites. L'entropie du LCG Faible (7.70 bits) est significativement éloignée de l'optimum (8.0), ce qui confirme une perte d'information et une prédictibilité élevée.

4.2.2 Le paradoxe du Mersenne Twister

Le Mersenne Twister obtient le meilleur score global (0.999) et une entropie excellente. Il réussit le test du Chi-2.

Conclusion critique : D'un point de vue purement statistique (répartition des nombres), c'est un excellent générateur. Cependant, comme démontré dans la section "Attaques", cette perfection statistique masque une insécurité cryptographique totale. C'est le piège classique : *"Cela ressemble à du bruit, donc c'est sûr"* est une hypothèse fausse.

4.2.3 La robustesse du SystemRNG et BBS

Le `System RNG` (basé sur `os.urandom`) affiche l'entropie la plus élevée (7.9970), quasiment parfaite. Sa P-value au Chi-2 (0.98) est très haute, indiquant une uniformité exemplaire. Le `Blum-Blum-Shub`, bien que beaucoup plus lent, valide également tous les critères d'uniformité, confirmant sa nature de CSPRNG théoriquement prouvé.

4.2.4 Conclusion de l'analyse

Les tests statistiques agissent comme un filtre nécessaire mais non suffisant.

1. Un générateur qui échoue ici (LCG) est à bannir immédiatement.
2. Un générateur qui réussit (Mersenne Twister) peut encore être vulnérable aux attaques par clonage d'état.
3. Seuls les générateurs utilisant des sources d'entropie externe (SystemRNG) ou des problèmes mathématiques durs (BBS) offrent à la fois qualité statistique et sécurité.

5 Démonstrations d'Attaques (Cryptanalyse)

Cette section documente les attaques réalisées sur les générateurs LCG et Mersenne Twister. L'objectif est de prédire les valeurs futures à partir d'observations passées.

5.1 Expérience 1 : Cassage algébrique du LCG

Nous montrons ici comment retrouver les paramètres secrets d'un LCG (multiplicateur a, incrément c) simplement en observant la sortie.

1. Modèle de menace

L'attaquant est passif (écoute seule). Il observe une séquence de nombres bruts X_1, X_2, X_3 générés par la cible. Il connaît le module m (souvent une puissance de 2 comme 2^{31}), mais ignore la clé interne (graine).

2. Hypothèses

— L'attaquant dispose d'au moins 3 sorties consécutives non tronquées.
— L'attaquant connaît le module m.

3. Algorithme d'attaque

Le système est régi par $X_{n+1} \equiv aX_n + c \pmod{m}$. Nous posons le système d'équations pour deux transitions successives :

$$\begin{aligned} X_1 &\equiv aX_0 + c \pmod{m} \\ X_2 &\equiv aX_1 + c \pmod{m} \end{aligned}$$

Par soustraction $(L_2 - L_1)$, nous éliminons l'inconnue c :

$$X_2 - X_1 \equiv a(X_1 - X_0) \pmod{m}$$

L'attaquant résout pour a en calculant l'inverse modulaire de $(X_1 - X_0)$:

$$a = (X_2 - X_1) \cdot (X_1 - X_0)^{-1} \pmod{m}$$

Une fois a connu, c est trivialement retrouvé : $c = X_1 - aX_0 \pmod{m}$.

4. Conditions de succès

L'attaque réussit si $(X_1 - X_0)$ est inversible modulo m, c'est-à-dire si $\text{PGCD}(X_1 - X_0, m) = 1$. Cette condition est statistiquement très fréquente.

5. Métriques et Résultats

— **Complexité :** $\mathcal{O}(1)$ (résolution instantanée).
— **Données nécessaires :** 3 entiers (12 octets).
— **Résultat obtenu :** Les paramètres a et c ont été récupérés. Nous avons pu instancier un "générateur clone" qui prédit 100% des valeurs suivantes de la victime.

5.2 Expérience 2 : Clonage d'état du Mersenne Twister

Cette attaque vise le générateur par défaut de Python (`random`). Elle ne cherche pas les paramètres (qui sont publics), mais l'**état interne** du générateur.

1. Modèle de menace

L'attaquant accède à une suite de valeurs générées par l'application (ex : ID de session, tokens de réinitialisation de mot de passe) et souhaite prédire les prochains tokens.

2. Hypothèses

— L'attaquant peut collecter 624 sorties consécutives de 32 bits.
— Les sorties correspondent directement à la sortie de la fonction `extract_number()` sans post-traitement complexe (comme un modulo strict).

3. Algorithme d'attaque (Untempering)

Le Mersenne Twister maintient un état de 624 entiers. Lorsqu'il produit un nombre, il prend une valeur de l'état et lui applique une transformation bijective appelée *Tempering* (mélange de bits via XOR et décalages) :

```
y = y ^ (y >> 11)
y = y ^ ((y << 7) & 0x9d2c5680)
# ... etc
```

Puisque ces opérations sont inversibles, nous avons implémenté une fonction `untemper(y)` qui inverse chaque opération bit-à-bit. En appliquant `untemper` sur 624 sorties observées, l'attaquant reconstruit l'intégralité du vecteur d'état interne au moment t.

4. Conditions de succès

Il est impératif d'avoir 624 valeurs contiguës. Si des valeurs manquent, l'attaque nécessite des techniques plus lourdes (SAT Solvers / Z3).

5. Métriques et Résultats

— **Données nécessaires :** 624 entiers de 32 bits (soit 2.5 Ko de données).

— **Résultat obtenu :** État interne entièrement cloné.
— **Preuve :** Le script d'attaque a prédit avec succès les 1000 valeurs suivantes générées par la cible. Le générateur est totalement compromis.

6 Conclusion et Recommandations

Ce projet met en lumière la dangerosité des générateurs pseudo-aléatoires classiques (PRNG) dans un contexte de sécurité.

1. **Distinction critique :** Un générateur peut produire une distribution uniforme parfaite (Mersenne Twister) tout en étant totalement prédictible. La qualité statistique n'implique pas la sécurité.
2. **Vulnérabilité :** Les algorithmes comme le LCG ou le Mersenne Twister permettent à un attaquant de cloner l'état du système avec très peu de données, rendant prévisibles toutes les clés ou jetons futurs.
3. **Recommandation finale :** Pour toute opération liée à la sécurité (authentification, cryptographie, jetons), il est impératif de bannir le module `random` et d'utiliser exclusivement les CSPRNG fournis par l'OS via le module `secrets` en Python (ou `System.Security.Cryptography` en C#).