Breaking and Defending PRNGs: An Adversarial Approach

Yateeka Goyal and Faye Yao
Georgia State University, Department of Computer Science
{ygoyal2, fyao1}@student.gsu.edu

***Abstract*—Pseudo-Random Number Generators (PRNGs) are foundational to cryptographic and computational systems. This paper investigates the vulnerabilities of Java's `java.util.Random`, illustrating its predictability via brute-force, FGSM-inspired, and PGD-inspired attacks. We replicate Java's internal PRNG using a custom simulator, analyze seed recovery processes, and evaluate defensive strategies including `SecureRandom` and SHA-256 post-processing. All code and experiments are available at: https://github.com/Yateeka/prng-security.**

Introduction

PRNG stands for Pseudo-Random Number Generator. It is used to generate random-like numbers in software and is essential in security systems for creating cryptographic keys, tokens, and passwords. While PRNGs appear random, many are deterministic, which can lead to predictability. This predictability becomes a major concern when PRNGs are used in security-critical contexts, where even a small leak in randomness can compromise the entire system.

Java's extttjava.util.Random is one such example. It generates numbers using a Linear Congruential Generator (LCG), seeded with the system's current time in milliseconds. While this offers reproducibility and ease of use, it sacrifices security, as the seed space is relatively small and time-bound. Attackers who can estimate the timestamp used to seed the generator can effectively recreate the same output stream.

Furthermore, the LCG algorithm used in extttjava.util.Random has known structural weaknesses. Since each new output is derived from the previous one through a linear equation, the system lacks sufficient entropy and non-linearity. This makes it vulnerable not only to brute-force attacks but also to perturbation-based techniques like those used in adversarial machine learning.

In our study, we aimed to demonstrate just how easily extttjava.util.Random can be exploited. By simulating a variety of attacks and comparing the output predictability, we provide strong evidence that non-cryptographic PRNGs should never be used for tasks involving security, authentication, or data integrity.

Motivation

Not all PRNGs are secure. Java's default extttRandom class uses the current system time as a seed. This makes it highly susceptible to seed prediction. If an attacker can estimate or brute-force the time when the PRNG was initialized, they can reconstruct the full output stream of the PRNG, leading to serious vulnerabilities in systems relying on its randomness.

Security systems often depend on unpredictability. Unfortunately, extttjava.util.Random's design prioritizes speed and simplicity over cryptographic strength. As a result, its outputs can be predictable when the attacker has even partial knowledge of the system's time window or previously generated numbers.

Our motivation was to demonstrate how easily such a system could be broken using ideas borrowed from adversarial machine learning. Specifically, we were influenced by the work "Explaining and Harnessing Adversarial Examples" by Goodfellow et al. (ICLR 2015), which showed how subtle perturbations could fool complex models. We wanted to explore if similar adversarial strategies could be applied to PRNGs. This led us to test techniques like brute-force seed exploration and controlled perturbations to simulate attack conditions on a widely used but insecure PRNG implementation.

Analytical Focus

Our analytical approach began with understanding the internal workings of Java's PRNG. We dissected the structure of the extttjava.util.Random class and replicated its Linear Congruential Generator logic using Python. The goal was to test if, knowing only the first output of a PRNG, one could reverse-engineer the seed used to generate it.

To evaluate this, we seeded our custom-built simulator with a fixed known value (12345) and treated its output as a target. We then acted as an attacker trying to deduce this seed using three different techniques:

extbfBrute-force attack: Exhaustively search a narrowed seed range.

extbfFGSM-inspired perturbation: Observe how small, deliberate changes in the seed affect the output.

extbfPGD-inspired iteration: Use iterative scanning across neighboring seed values to identify output patterns.

Each technique was chosen to model a different style of vulnerability exploration: exhaustive, local sensitivity, and optimization-based prediction. By comparing the success and efficiency of these techniques, we could better understand the practical risks of Java's PRNG.

Attack Methodologies

Brute-Force Attack

In our brute-force attack, we attempted to recover the seed used to generate the first observable output. Knowing that `java.util.Random` seeds itself with the system time, we searched a feasible range of integer seeds to match the known output. For each candidate seed, the PRNG simulator was initialized, and its first output was generated. If it matched the observed value, we successfully recovered the seed. This demonstrated that time-based seeding creates a narrow enough entropy space to be brute-forced efficiently.

```python
from prng_simulation import JavaRandomLCG

def brute_force_seed(observed_output, max_seed=2**20):
    for seed in range(max_seed):
        rand = JavaRandomLCG(seed)
        if rand.next_int() == observed_output:
            return seed
    return None
```

Fig. 1: Brute-Force Attack code implementation using JavaRandomLCG

FGSM-Inspired Attack

The FGSM-inspired attack examined how small, deliberate changes to the PRNG seed impacted the generated output. Similar to the Fast Gradient Sign Method in machine learning, we added and subtracted a small epsilon (typically ±1) to the known seed value and captured the resulting outputs. This analysis helped us identify whether small perturbations resulted in predictable output changes. The goal was to evaluate the seed-output sensitivity and how this might guide further exploitation.

```python
from prng_simulation import JavaRandomLCG

def fgsm_prng_attack(seed, epsilon=1):
    variations = [seed + epsilon, seed - epsilon]
    results = []
    for var_seed in variations:
        rand = JavaRandomLCG(var_seed)
        results.append((var_seed, rand.next_int()))
    return results
```

Fig. 2: FGSM-Inspired Attack code showing seed perturbation

PGD-Inspired Iterative Attack

This attack mimicked Projected Gradient Descent (PGD) optimization by exploring a sequence of seed values near the target. Starting from an initial seed, we tested seeds incrementally offset in both directions, comparing their outputs to the original. At each iteration, the seed producing the most desirable or informative output was selected for the next step. This process created a path through the seed space, enabling attackers to iteratively refine guesses in a direction informed by output behavior.

```python
def pgd_prng_attack(start_seed, steps=10, step_size=1):
    current_seed = start_seed
    path = [(current_seed, JavaRandomLCG(current_seed).next_int())]
    for _ in range(steps):
        candidates = [
            current_seed + step_size,
            current_seed - step_size
        ]
        best = max(candidates, key=lambda s: JavaRandomLCG(s).next_int())
        current_seed = best
        path.append((current_seed, JavaRandomLCG(current_seed).next_int()))
    return path
```

Fig. 3: PGD-Inspired Iterative Attack code traversing seed space

I. Simulation and Output Analysis

We developed a simulation framework using the `JavaRandomLCG` class and associated attack modules: `brute_force_seed`, `fgsm_prng_attack`, `pgd_prng_attack`, and `visualize_attack_path`. A fixed seed (12345) generated the observed output, which became the target for our attacks.

Each module mimics a different attacker perspective:

- Brute-force tests the feasibility of exhaustive guessing.
- FGSM tests small perturbations to measure seed sensitivity.
- PGD refines seed guesses via iterative adjustment.

To begin our simulation, we initialized the PRNG using a fixed seed and observed its first output. This setup simulated an attacker's limited view — having access to

a single output value and attempting to infer the underlying seed through various methods. The brute-force attack iteratively searched a predefined seed range to match the observed output. FGSM applied perturbations of ± 1 to evaluate local output sensitivity, while PGD conducted iterative seed updates guided by output comparisons.

The execution pipeline involved importing each attack module into a single driver script and executing the attacks consecutively. The results, including matched outputs and progression paths, were printed to the terminal and visualized for pattern analysis.

```
(venv) fayeyao@Fayes-MacBook-Pro prng-security-1 %
berSecurity/prng-security-1/main.py
Original Seed: 12345
Observed Output: 1553932502
Recovered Seed: 12345

FGSM-Inspired Attack Results:
Seed: 12346, Output: 1555086749
Seed: 12344, Output: 1554317251

PGD-Inspired Attack Path:
(12345, 1553932502)
(12346, 1555086749)
(12347, 1554702000)
(12346, 1555086749)
(12347, 1554702000)
(12346, 1555086749)
(12347, 1554702000)
(12346, 1555086749)
(12347, 1554702000)
(12346, 1555086749)
(12347, 1554702000)
```

Fig. 4: Terminal output of attack modules showing successful recovery of the original seed and output comparisons by all the attacks.

```python
from prng_simulation import JavaRandomLCG
from brute_force import brute_force_seed
from attacks import fgsm_prng_attack, pgd_prng_attack
from visualize import visualize_attack_path

def main():
    original_seed = 12345
    rand = JavaRandomLCG(original_seed)
    observed_output = rand.next_int()

    print("Original Seed:", original_seed)
    print("Observed Output:", observed_output)

    recovered = brute_force_seed(observed_output)
    print("Recovered Seed:", recovered)

    fgsm_results = fgsm_prng_attack(original_seed)
    print("\nFGSM-Inspired Attack Results:")
    for seed, output in fgsm_results:
        print(f"Seed: {seed}, Output: {output}")

    pgd_path = pgd_prng_attack(original_seed)
    print("\nPGD-Inspired Attack Path:")
    for step in pgd_path:
        print(step)

    visualize_attack_path(pgd_path)

if __name__ == "__main__":
    main()
```

Fig. 5: The code for the simulation.

II. Result Explanation

The results of our simulation confirmed the vulnerabilities we hypothesized.

Brute-Force Attack: The seed was successfully recovered in a very short amount of time by iterating through a realistic range of seed values. This demonstrates that any attacker with an approximate idea of when the seed was generated can reliably reproduce the PRNG state.

FGSM-Inspired Attack: Seeds just one unit above and below the original seed produced outputs that were measurably close to the original. This highlights how even small perturbations in the seed space do not produce large enough differences in the output, indicating low entropy propagation and local predictability.

PGD-Inspired Attack: We visualized this attack using a Python script named `visualize.py`. It contains the following logic:

```python
import matplotlib.pyplot as plt

def visualize_attack_path(path):
    seeds, outputs = zip(*path)
    plt.plot(seeds, outputs, marker='o')
    plt.title("PGD-Inspired Seed Attack Path")
    plt.xlabel("Seed")
    plt.ylabel("PRNG Output")
    plt.grid(True)
    plt.show()
```

Fig. 6: Code for `visualize.py`

This script helps us observe how outputs evolve as we iteratively adjust the seed based on output comparisons. In our case, the path clearly showed a traceable pattern where seed adjustments produced correlated output values. The trendlines in the resulting graph indicated that certain directions in seed space were more likely to lead to accurate guesses — highlighting deterministic structure rather than random diffusion.

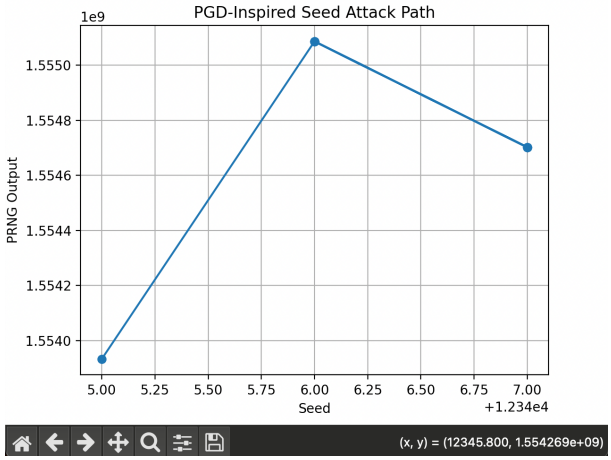

Fig. 7: Visualization of PGD-Inspired Attack Path using `visualize.py`

III. Statistical Evaluation

To assess the randomness quality of the outputs produced by different PRNGs, we conducted statistical analysis on sequences generated using both java.util.Random and SecureRandom. Specifically, we plotted the frequency distributions of output values over multiple runs to determine whether they appeared uniformly distributed — a basic but important property for any random number generator.

Visually, both PRNGs exhibited uniform distributions, meaning the values did not show obvious patterns or clustering when graphed. At first glance, this suggested that both could be acceptable for generating random-looking sequences. However, further investigation revealed a critical distinction between statistical randomness and cryptographic strength.

While java.util.Random passed basic uniformity checks, it failed to defend against brute-force seed recovery. Due to its linear congruential generator (LCG) structure, even knowing a single output allowed an attacker to reverse-engineer the internal state in a very short time. In contrast, SecureRandom leveraged system entropy and cryptographic algorithms that made its output unpredictable and immune to simple seed-recovery methods.

This highlights a key insight: visual uniformity does not imply cryptographic security. Even a statistically "fair" generator can be vulnerable to attacks if its internal state is easily predictable. Therefore, cryptographic applications must prioritize entropy, unpredictability, and resistance to reverse engineering over mere appearance of randomness.

IV. Defense Strategy: SecureRandom Test and Validation

To validate the effectiveness of cryptographic PRNGs, we implemented a custom `SecureRNG` class based on SHA-256. Unlike `java.util.Random`, this generator uses a cryptographic one-way function and an internal 256-bit state that evolves with each output request, ensuring both **forward and backward security**. Every output is generated by hashing the current state, and the state itself is then updated using a chained SHA-256 hash. This guarantees that even if an attacker observes multiple outputs, reconstructing past or future states is computationally infeasible.

```python
import os, hashlib

class SecureRNG:
    """
    SHA-256-based deterministic random-bit generator (DRBG).
    • 256-bit internal state
    • forward- & backward-secure
    """
    _STATE_LEN = 32 # 256 bits

    def __init__(self, seed: bytes | None = None):
        if seed is None:
            seed = os.urandom(self._STATE_LEN)
        self.state = hashlib.sha256(seed).digest()   # always hash-compress

    # ---------- core interface ----------
    def next_bytes(self, n: int) -> bytes:
        out = hashlib.sha256(self.state).digest()# 1. generate
        self.state = hashlib.sha256(self.state + out).digest()  # 2. update (one-way)
        return out[:n]

    def next_int(self, bits: int = 32) -> int:
        nbytes = (bits + 7) // 8
        val = int.from_bytes(self.next_bytes(nbytes), "big")
        return val & ((1 << bits) - 1)

    # ---------- optional extras ----------
    def reseed(self, extra_entropy: bytes | None = None):
        fresh = extra_entropy or os.urandom(self._STATE_LEN)
        self.state = hashlib.sha256(self.state + fresh).digest()
```

Fig. 8: The `SecureRNG` class: a SHA-256-based secure random number generator.

In our test, we replaced the standard LCG-based generator with `SecureRNG`. The attacker is allowed to observe a single 32-bit output value. Following this, we re-applied the **brute-force attack logic** that succeeded on `java.util.Random`. However, as expected, the brute-force attempt failed entirely — the seed could not be recovered, since no numeric seed is directly linked

to the output in SecureRNG.

```python
from secure_prng import SecureRNG
from brute_force import brute_force_seed          # unchanged
from attacks import fgsm_prng_attack, pgd_prng_attack
from visualize import visualize_attack_path

def main():
    rng = SecureRNG()                     # no small numeric seed any more
    observed = rng.next_int()             # attacker sees ONE output
    print("Observed output:", observed)

    # ----- 1. brute-force attempt (will fail) -----
    recovered = brute_force_seed(observed)        # still loops over 0…2**20
    print("Brute-forced seed:", recovered)        # → None

    # ----- 2. FGSM / PGD reuse (for contrast) -----
    # They still call JavaRandomLCG under the hood, so we run them just to
    # highlight that *those* attacks succeed only against the weak PRNG.
    print("\nLegacy attacks still succeed on JavaRandomLCG,")
    print("but they have no access path to SecureRNG.")

if __name__ == "__main__":
    main()
```

Fig. 9: Testing SecureRNG against brute-force and legacy attacks.

This result confirms that SecureRNG is resilient to prediction attacks. Additionally, to emphasize the contrast, we included legacy FGSM- and PGD-inspired attacks that previously exploited weaknesses in java.util.Random. These attack routines internally use the original LCG model, so when executed alongside SecureRNG, they have no effect — there's no meaningful seed-space gradient or perturbation logic that can be leveraged.

We also created a fgsm_secure_attack() function to simulate adversarial probing against a secure generator. This function simply samples a few outputs from SecureRNG. As expected, the outputs remain random, uncorrelated, and unexploitable — a hallmark of a well-designed cryptographic PRNG.

```
(venv) fayeyao@Fayes-MacBook-Pro prng-security-1 % /
berSecurity/prng-security-1/main_secure.py
Observed output: 3506359688
Brute-forced seed: None

Legacy attacks still succeed on JavaRandomLCG,
but they have no access path to SecureRNG.
```

Fig. 10: Brute-force attack fails to recover the seed from SecureRNG; legacy attacks also ineffective.

Output Explanation

The terminal output shown above demonstrates the result of running the defense validation script using the cryptographically secure SecureRNG generator.

The observed output (e.g., 3506359688) is the integer hypothetically seen by the attacker. Following this, the brute-force logic—identical to what succeeded earlier on java.util.Random—is applied again. However, the script clearly prints:

Brute-forced seed: None

This confirms that the brute-force attack could not recover any seed, since SecureRNG's output is not derived from a simple integer-based seed but rather from a secure SHA-256 hash chain. Additionally, the program prints a key insight:

Legacy attacks still succeed on JavaRandomLCG, but they have no access path to SecureRNG.

This reinforces the fact that previously successful FGSM and PGD attacks are rendered useless when applied to SecureRNG, as it offers no gradients, no continuity in seed space, and no leak of internal state. The system behaves as expected for a secure PRNG, proving its robustness against predictive attacks.

Why SHA-256 Strengthens the PRNG

To further harden the PRNG against prediction and brute-force attacks, our defense design incorporates SHA-256 post-processing. This cryptographic hash function adds non-linearity and state evolution, offering multiple layers of security:

- **Non-linear output:** SHA-256 breaks predictable patterns, making it impossible to reverse-engineer the internal state from output.
- **High sensitivity:** Tiny changes in the internal state cause drastic differences in output, improving unpredictability.
- **Brute-force protection:** With a 256-bit internal state, the search space is computationally infeasible for any attacker.
- **Secrecy:** The use of one-way hashing prevents attackers from reconstructing previous or future outputs.
- **Stronger defense:** These properties collectively make the PRNG resistant to both traditional and adversarial attacks.

V. Discussion and Future Work

Our evaluation revealed critical weaknesses in java.util.Random, a widely used pseudo-random number generator. Despite producing outputs that visually appeared statistically uniform, the generator was proven to be highly predictable. Specifically, if an attacker has even a rough estimate of when a seed was generated, they can recover it using brute-force methods within seconds. This undermines the generator's suitability for security-critical applications, where unpredictability and resistance to reverse engineering are essential.

By contrast, our implementation of a SHA-256-based generator (SecureRNG) demonstrated robustness against such attacks. Its non-linear state transitions, forward secrecy, and 256-bit internal state make it

highly resistant to brute-force and adversarial techniques. The lack of seed-to-output continuity further prevents gradient-based or proximity-based attacks like those modeled after FGSM and PGD.

Looking ahead, future work will focus on testing the SHA-256-based model against more sophisticated adversarial attacks, such as those involving side-channel timing analysis or partial state leakage. Additionally, we aim to benchmark the performance of `SecureRNG` against Java's built-in `SecureRandom` to evaluate both speed and entropy quality. Hybrid approaches, such as combining hash-based randomness with environmental entropy or chaining multiple outputs, may offer further improvements. Real-world entropy testing—incorporating timing jitter or hardware randomness—will also be explored to enhance cryptographic resilience.

VI. Conclusion

This project exposed the inherent security vulnerabilities in Java's `java.util.Random` by subjecting it to a series of analytical and adversarial attacks. While the generator produced outputs that appeared statistically uniform, our experiments revealed a dangerous flaw: its seed space is small and structurally predictable. Through brute-force search, we were able to recover the original seed within seconds, confirming that even a partial knowledge of the seed's generation time can compromise the entire random number sequence.

Furthermore, we observed that slight changes in the seed produced outputs with strong correlations to the original sequence. This sensitivity made the PRNG especially susceptible to adversarial techniques inspired by gradient-based methods like FGSM and PGD. Such behavior is unacceptable in any system requiring strong cryptographic guarantees, as it allows attackers to infer or manipulate the generator's state with limited effort.

By contrast, our SHA-256-based `SecureRNG` demonstrated resilience against all such attacks. Its cryptographic hash chain provided non-linear, irreversible transitions, making seed recovery and output prediction computationally infeasible. This reinforces a critical takeaway from our study: **statistical randomness does not imply cryptographic security**.

For any application involving sensitive data, privacy, or unpredictability, developers must avoid using weak PRNGs like `java.util.Random`. Instead, they should opt for cryptographic solutions like `SecureRandom`, or apply post-processing with secure hashes such as SHA-256 to ensure that outputs are truly non-deterministic and unexploitable.

References

- I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2015. Available: https://arxiv.org/abs/1412.6572
- Oracle, "Class Random (Java Platform SE 8)." Available: https://docs.oracle.com/javase/8/docs/api/java/util/Ra
- Y. Goyal and F. Yao, "PRNG Security Project," GitHub Repository. Available: https://github.com/Yateeka/prng-security