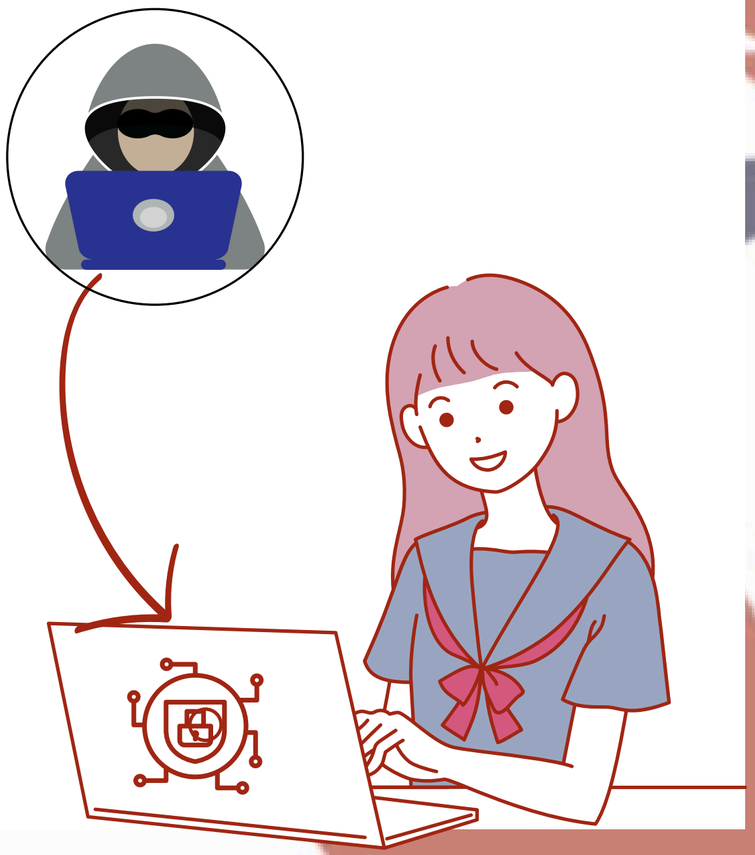
Breaking and Defending PRNGs: An Adversarial Approach

Enhancing PRNG Security with SHA-256: A Defense Against Seed Prediction

By: Yateeka Goyal, Faye Yao

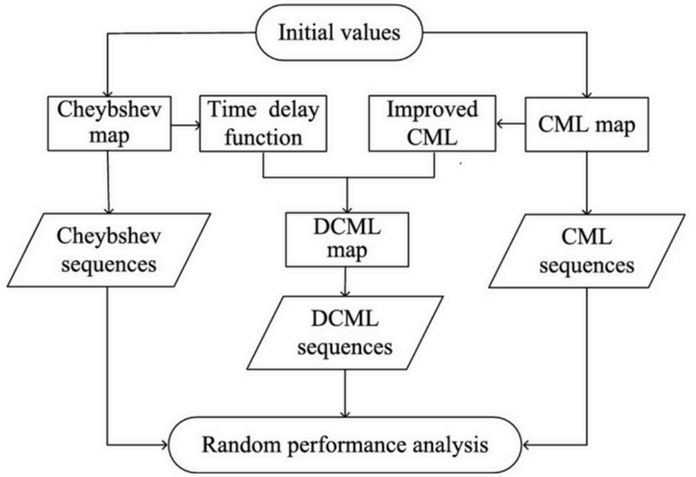
Agenda

1. Introduction
2. Motivation
3. Analytical Focus
4. Attack Methodologies
5. Simulation & Output Analysis
6. Statistical Evaluation
7. PRNG Defense
8. Discussion & Future Work
9. Conclusion

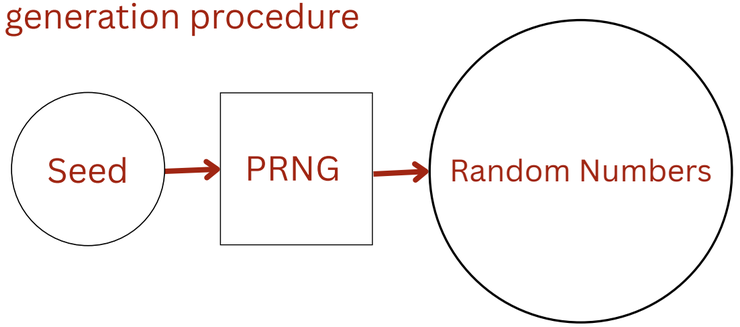


Introduction

- PRNG stands for Pseudo-Random Number Generator.
- Commonly used for generating random-like numbers in software.
- Essential in security systems: password generation, tokens, cryptographic keys.
- But not all PRNGs are secure — especially if they are predictable



A flowchart of pseudo-random number generation procedure



Motivation

- In cybersecurity, a robust PRNG is crucial: if its “random” output can be predicted, attackers can recreate cryptographic keys and tokens, undermining encryption and exposing entire systems.
- Java’s PRNG uses current time as a seed. Attackers can predict “random” numbers if they know the seed.
- Real-world implications: token hijacking, password leaks, etc.

```
contract UnsafeDependenceOnBlock {
    uint8 answer;
    function UnsafeDependenceOnBlock() public payable {
        require(msg.value == 1 ether);
        answer =
            uint8(keccak256(block.blockhash(block.number - 1), now));
    }
    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }
    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);
        if (n == answer) {
            msg.sender.transfer(2 ether);
        }
    }
}
```

An example of weak randomness.

Analytical Focus

To demonstrate the vulnerability of Java's PRNG (java.util.Random) through seed recovery and adversarial attack simulations, inspired by ***"Explaining and Harnessing Adversarial Examples" by Goodfellow, Shlens, and Szegedy (ICLR 2015)***.

Experimental Setup:

- Build a custom PRNG simulator replicating Java's PRNG.
- Initialize a known seed and note its output.
- Pretend to be an attacker trying to guess that seed and predict the next numbers.

Attack Methodologies

- Brute-Force Attack:
  - Recovered the original seed by testing seeds over a small time-based range.

- FGSM-Inspired Method:
  - Tested neighboring seeds (seed ± 1) to evaluate sensitivity of the output.

- PGD-Inspired Method:
  - Kept adjusting the seed (+1 / -1) step by step to see how each change affects the random number.

Simulation & Output Analysis

- JavaRandomLCG Class — PRNG Simulator
  - What it does:
    - Mimics Java's java.util.Random logic using Linear Congruential Generator (LCG).
    - Core class for generating outputs based on a known seed.

```python
class JavaRandomLCG:
    def __init__(self, seed):
        self.seed = (seed ^ 0x5DEECE66D) & ((1 << 48) - 1)
        self.multiplier = 0x5DEECE66D
        self.addend = 0xB
        self.mask = (1 << 48) - 1

    def next(self, bits):
        self.seed = (self.seed * self.multiplier + self.addend) & self.mask
        return self.seed >> (48 - bits)

    def next_int(self):
        return self.next(32)
```

Simulation & Output Analysis

- Victim Simulation — Generating a Target Output
  - What it does:
    - Initializes a PRNG with a seed (e.g., 12345).
    - Produces the first output — this is the value seen by the attacker

```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()
```

Simulation & Output Analysis

- Brute-Force Seed Recovery
  - What it does:
    - Loops through a range of possible seed values.
    - Compares each output to the known observed value.
    - If a match is found, the correct seed is recovered.

```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
```

```
berSecurity/prng-security-1/main.py
Original Seed: 12345
Observed Output: 1553932502
Recovered Seed: 12345
```

Simulation & Output Analysis

- FGSM(Fast Gradient Sign Method) Attack — Seed ± 1 Test
  - What it does:
    - Bump the seed up by 1 or down by 1.
    - Note how the random number changes.
    - Large changes mean Java's generator is very ***sensitive*** to tiny seed tweaks.

```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
```

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

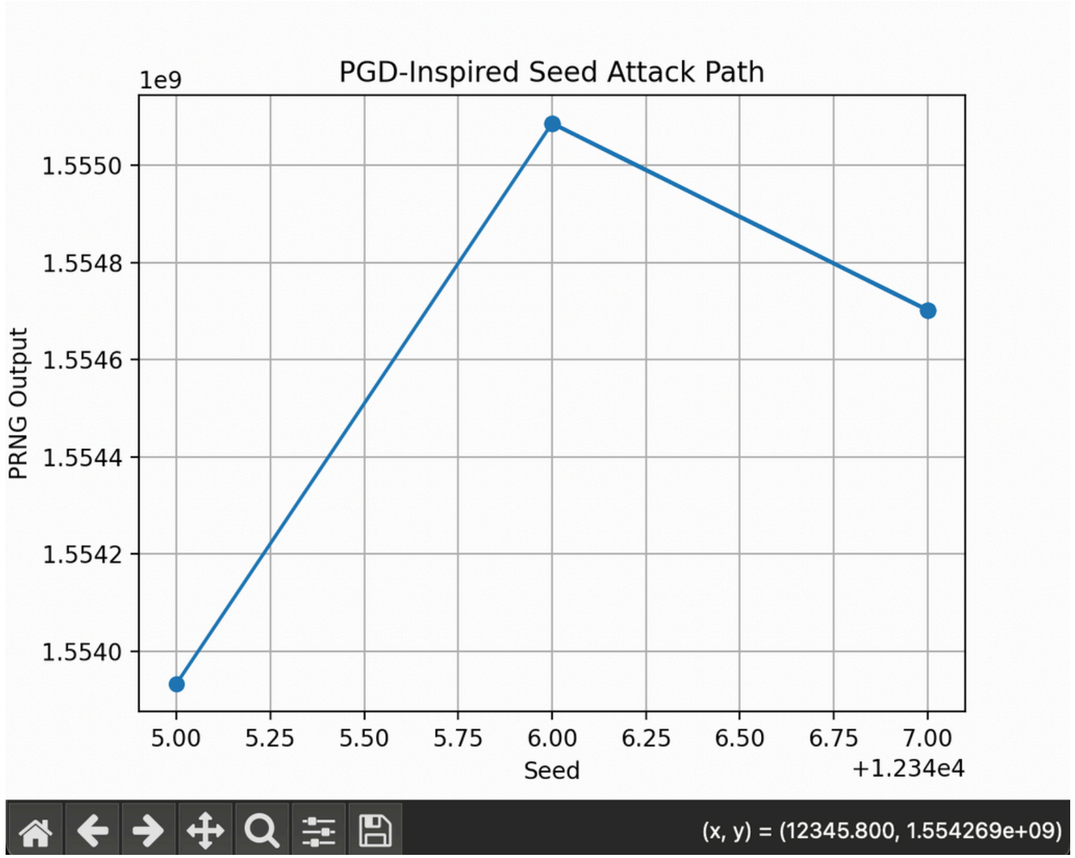
Simulation & Output Analysis

- PGD(Projected GradientDescent) Attack
  - What it does:
    - Try the seeds above and below the current one. (Seed ± 1)
    - Jump to the seed that gives the bigger random number, then **repeat**, **log** and **trace** the path.
    - Shows how the attacker can manipulate seeds to produce more **disered** outputs.

```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
```

Simulation & Output Analysis

```
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)
```



Simulation & Output Analysis

- SecureRandom Test — Defense Validation
  - What it does:
    - Uses SecureRandom to generate a secure output.
    - Brute-force logic is applied again — fails to recover seed.
    - Proves SecureRandom is resistant to such prediction attacks.

```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()
```

```
(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.
```

Statistical Evaluation

We analyzed outputs from both **java.util.Random** and **SecureRandom**. Both showed uniform frequency distributions when plotted.

However, only SecureRandom resisted brute-force attacks. java.util.Random was statistically random but cryptographically weak. This confirms that **visual uniformity ≠ true security**.

PRNG Defense

- To improve strengthen PRNG, we used SHA-256 post-processing to increase output unpredictability and reduce attack success.

    - **Non-linear output**: SHA-256 breaks predictable patterns.
    - **High sensitivity**: Tiny state changes cause major output shifts.
    - **Brute-force protection**: 256-bit state is too large to search.
    - **Secrecy**: Past and future outputs can't be reconstructed.
    - **Stronger defense**: Makes PRNG resistant to adversarial attacks.

Discussion & Future Work

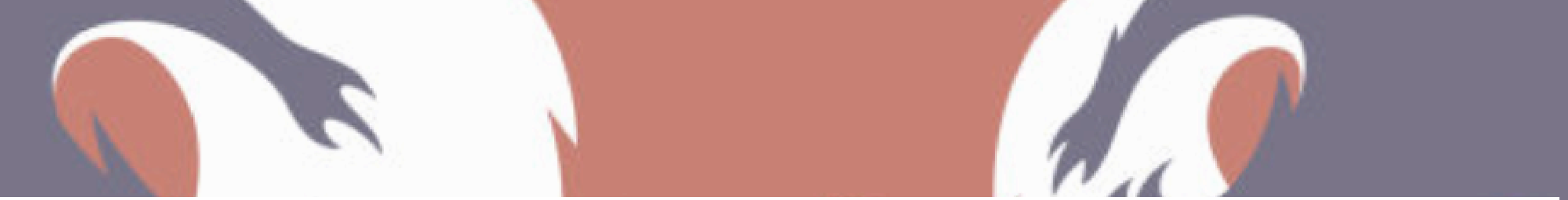
- Java’s PRNG is weak and predictable. If the seed is known, we can reproduce the exact same “random” numbers, so avoid using it on its own in security-sensitive applications.

- Future Work
  - Apply and test hash-function SHA-256 against advanced attacks.
  - Compare performance vs. SecureRandom.
  - Measure speed and add real-life randomness tests

Conclusion

We found that tiny adjustment to the seed allows attackers to predict and even recover the seed, so PRNG is too vulnerable for security purposes. Applying hash function on each "random" output with SHA-256 breaks those simple patterns and makes guessing the seed much more complicated.

Key takeaway: just because a PRNG looks random doesn’t mean it’s safe. When you need real unpredictability, use SecureRandom or a generator that’s been strengthened with SHA-256.



Thank you