COAL: Computer Organization and Assembly Language.

Project Report: **Yarrow Random Number Generator.**

**Name:** Alizeh Affif Khan, Maaz Saeed, Shahal Farid.

**Class:** CYS 4 A

**Roll no.:** 221590, 221492, 221484

1. **Introduction:**

The Yarrow Random Number Generator is a cryptographic pseudorandom number generator designed to produce high-quality random numbers for cryptographic applications. This project aims to implement the Yarrow algorithm in Python, providing a secure and reliable source of randomness.

1. **Features:**

**Entropy Pool:** Accumulates entropy from various sources, ensuring randomness.

**Reseeding:** Automatically reseeds the generator at specified intervals to maintain randomness.

**Thread Safety:** Utilizes threading locks to ensure safe concurrent access to shared resources.

**Configuration:** Allows dynamic configuration of reseed interval and maximum pool size.

**Entropy Sources:** Supports multiple entropy sources including system events, user input, and hardware RNG (placeholder).

1. **Implementation:**

**Initialization:** The Yarrow class is initialized with default or user-defined reseed interval and maximum pool size.

**Entropy Accumulation:** Entropy is added to the pool using the **add\_entropy** method, which extends the entropy pool and ensures it does not exceed the maximum size.

**Reseeding:** The generator is reseeded automatically based on the reseed interval to refresh the seed and maintain randomness.

**Random Number Generation:** Random bytes are generated using a secure PRNG based on SHA-256 hashing. The **generate\_random\_number** method produces random integers within a specified range.

**Entropy Quality Assessment:** Placeholder methods are provided for assessing the quality of entropy and authenticating entropy sources.

**Error Handling:** Errors are handled gracefully using the **handle\_errors** method.

**Logging:** Usage of the generator is logged using the **log\_usage** method.

1. **Testing and Integration:**

**Integration Testing:** Placeholder method **test\_integration** is provided for testing the integration of entropy sources with the Yarrow algorithm.

**Statistical Testing:** The **chi\_squared\_test** method performs a chi-squared test on the generated data to evaluate randomness.

1. **Future Improvements:**

Implement cryptographic signature verification for authenticating entropy sources.

Integrate hardware RNG support for additional entropy.

Enhance entropy quality assessment using NIST Statistical Test Suite or Diehard tests.

Develop comprehensive integration tests to validate entropy source integration.

1. **Conclusion:**

The Yarrow Random Number Generator provides a robust and secure solution for generating random numbers in cryptographic applications. With its dynamic reseeding mechanism, thread safety, and support for multiple entropy sources, it ensures high-quality randomness essential for cryptographic protocols and applications.

1. **Code:**

import os

import hashlib

import time

import threading

import random

from scipy.stats import chisquare

class Yarrow:

def \_\_init\_\_(self, reseed\_interval=10000, max\_pool\_size=4096):

"""

Initialize the Yarrow random number generator.

:param reseed\_interval: Reseed interval in bytes

:param max\_pool\_size: Maximum entropy pool size

"""

self.entropy\_pool = bytearray()

self.seed = bytearray()

self.reseed\_interval = reseed\_interval

self.max\_pool\_size = max\_pool\_size

self.last\_reseed\_time = time.time()

self.lock = threading.Lock()

def add\_entropy(self, data):

"""

Add entropy to the entropy pool.

:param data: Data to be added as entropy

"""

with self.lock:

if data:

self.entropy\_pool.extend(data)

if len(self.entropy\_pool) > self.max\_pool\_size:

self.entropy\_pool = self.entropy\_pool[-self.max\_pool\_size:]

else:

print("Warning: Empty entropy data received.")

def assess\_entropy\_quality(self):

"""

Assess the quality of entropy in the entropy pool.

Implement NIST Statistical Test Suite or Diehard tests.

"""

pass

def authenticate\_source(self, source):

"""

Authenticate the entropy source.

Implement cryptographic signature verification or authentication mechanisms.

"""

pass

def add\_entropy\_sources(self, sources):

"""

Add multiple entropy sources to increase diversity.

:param sources: List of entropy sources

"""

for source in sources:

self.add\_entropy(source)

def generate\_seed(self):

"""

Generate a new seed from the entropy pool.

"""

with self.lock:

self.seed = hashlib.sha256(self.entropy\_pool).digest()

def reseed(self):

"""

Reseed the generator if necessary.

"""

with self.lock:

if len(self.seed) >= self.reseed\_interval or time.time() - self.last\_reseed\_time >= self.reseed\_interval:

self.generate\_seed()

self.last\_reseed\_time = time.time()

def generate\_random\_bytes(self, n):

"""

Generate 'n' random bytes.

:param n: Number of random bytes to generate

"""

self.reseed()

prng = os.urandom(32)

random\_bytes = bytearray()

while len(random\_bytes) < n:

prng = hashlib.sha256(prng).digest()

random\_bytes.extend(prng)

return bytes(random\_bytes[:n])

def add\_system\_entropy(self):

"""

Add entropy from various system sources.

"""

self.add\_entropy(os.urandom(32))

self.add\_entropy(bytes(str(time.time()), 'utf-8'))

def add\_user\_entropy(self, user\_input):

"""

Add entropy based on user input.

:param user\_input: User input as a string

"""

self.add\_entropy(bytes(user\_input, 'utf-8'))

def add\_hardware\_entropy(self):

"""

Add entropy from hardware RNG if available.

"""

# Placeholder function for hardware RNG support

pass

def generate\_random\_number(self, low, high):

"""

Generate a random number between 'low' and 'high' (inclusive).

:param low: Lower bound of the random number

:param high: Upper bound of the random number

"""

range\_size = high - low + 1

num\_bytes = (range\_size.bit\_length() + 7) // 8 # Calculate number of bytes needed

random\_data = self.generate\_random\_bytes(num\_bytes)

random\_number = int.from\_bytes(random\_data, byteorder='big')

return low + random\_number % range\_size

def chi\_squared\_test(self, data):

"""

Perform a chi-squared test on the data.

:param data: Data to be tested

"""

observed = [data.count(x) for x in range(256)]

expected = [len(data) / 256] \* 256

chi\_stat, \_ = chisquare(observed, expected)

return chi\_stat

def log\_usage(self, action):

"""

Log the usage of the generator.

:param action: Action to be logged

"""

print(f"Logging: Action '{action}' logged.")

def test\_integration(self):

"""

Test integration of entropy sources with the Yarrow algorithm.

"""

# Implement integration tests here

pass

def handle\_errors(self, error):

"""

Handle errors gracefully.

:param error: Error message or exception

"""

print(f"Error: {error}")

def configure(self, reseed\_interval=None, max\_pool\_size=None):

"""

Configure the Yarrow generator.

:param reseed\_interval: Reseed interval in bytes

:param max\_pool\_size: Maximum entropy pool size

"""

with self.lock:

if reseed\_interval is not None:

self.reseed\_interval = reseed\_interval

if max\_pool\_size is not None:

self.max\_pool\_size = max\_pool\_size

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

rng = Yarrow()

# Configure parameters

rng.configure(reseed\_interval=20000, max\_pool\_size=8192)

# Generate 32 bytes of random data

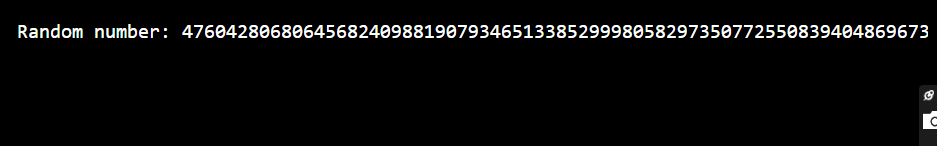
random\_data = rng.generate\_random\_bytes(32)

# Convert the random data to an integer

random\_number = int.from\_bytes(random\_data, byteorder='big')

print("Random number:", random\_number)

1. **Output:**

****