CRYPTOGRAPHY AND SECURITY

Practice

IP-18FKVKRBG

Lecture 5

2022-2023-2

More about PRNG

Random Number Generators (RNGs)

- TRNG — **True Random Number Generator**
- PRNG — **Pseudo Random Number Generator**
- CSPRNG — **Cryptographically Secure RNG**

Definition (PRNG)

$$G: \{0,1\}^k \rightarrow \{0,1\}^n$$

- with the following properties:
  - $n >> k$
  - G(x) computationally indistinguishable from true random
  - **the values are uniformly distributed over a defined interval**
  - it is impossible to predict future values based on past or present ones
  - there are no correlations between successive numbers

Probability Distribution

- A **probability distribution** is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment.

- We can pick randomly according to any "probability distribution":

$$P : S \rightarrow R$$

- The probability distribution P assigns a nonnegative probability to each $x \in S$, such that:

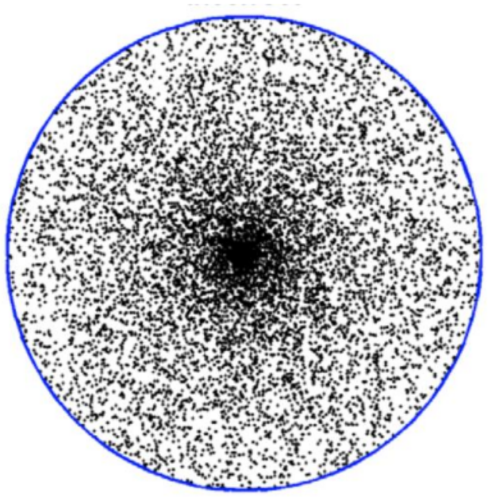
$$\sum_{x \in S} p(x) = 1$$

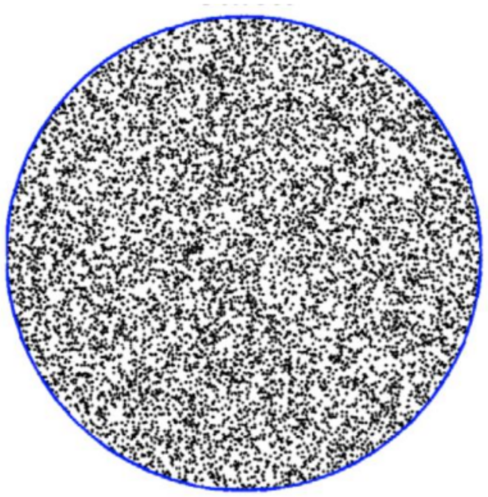
Uniform Distribution

- What if we want to pick something **at random**?

- The uniform distribution U is the probability distribution where everything is picked equally often:

$$If\ |S| = n, then\ \Pr[x = s] = \frac{1}{n}, \qquad \forall s \in_u S$$





Next bit test

A sequence of bits passes the next bit test for at any position i in the sequence, if:

- *an adversary knows the i first bits, and*
- *He cannot predict the* $(i + 1)$: *with probability of success better than* 50% + ε.

CSPRNG

Criteria for Cryptographically-Secure Pseudo random number generators (CSPRNG):

- pass statistical randomness tests
- resist against well-known cryptographic attacks
- Every CSPRNG should satisfy the next-bit test.

- ***Yao's theorem:*** *If a generator passing the next-bit test will pass all other polynomial-time statistical tests for randomness.*

Linear Congruential Generators

Linear Congruential Generators

LCG generators are defined by the following recursive formula:

$$\boldsymbol{X_{n+1} \equiv (a\,X_n + c)(mod\; m)}$$

Where:

- a is the multiplier ($0 < a < m$)
- X_0 is a starting seed value
- c is the increment value ($0 \leq c < m$)
- $m > 0$ is the modulus
- It is known that the period of a general LCG is at most m.

Linear Congruential Generators

Linear congruential generators (LCG) are one of the best known pseudorandom number generators. Few example without loss of generality:

- ANSI C, C++
- Microsoft Visual/Quick C/C++
- Microsoft Visual Basic (6 and earlier)
- Java’s java.util.Random
- Turbo Pascal, Borladn Delphi
- Apple CarbonLib
- C++11’s minstd rand

Linear Congruential Generators

State n = multiplier State n-1 + increment **mod** modulus

```
def prng_lcg(seed, repeat):
    multiplier = 3170695042276722 57  # the "multiplier"
    increment = 1035085024576065627  # the "increment"
    modulus = 8512677386048191063  # the "modulus"
    state = seed
    file= open("LCG.rnd","w")

    for _ in range(repeat):
        state = (state * multiplier + increment) % modulus
        file.write(str(state).zfill(19)+"\n")

prng_lcg(300, 300)
```

LCG Cracking: Unknown increment

State n = multiplier State n-1 + ? **mod** modulus

? = State n - multiplier State n-1 **mod** modulus

LCG Cracking: Unknown increment

```
modulus = 8512677386048191063
multiplier = 317069504227672257
increment = 1035085024576065627
rn = [0, 0, 0, 0, 0, 0]
file = open("LCG.rnd","r")
for i in range(6):
    rn[i] = int(file.readline())
```

```
def crack_unknown_increment(states, modulus, multiplier):
    increment = (states[1] - states[0]*multiplier) % modulus
    return increment
```

```
recovered_increment = crack_unknown_increment(rn, modulus, multiplier)
```

LCG Cracking: Unknown multiplier

State n-1 = ? State n-2 + increment **mod** modulus

State n = ? State n-1 + increment **mod** modulus

multiplier = State n - State n-1 / State n-1 - State n-2 **mod** modulus

LCG Cracking: Unknown multiplier

```python
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)

def modinv(b, n):  # Modular inverse function
    g, x, _ = egcd(b, n)
    if g == 1:
        return x % n
```

```python
def gcd(x, y): # GCD
    while(y):
        x, y = y, x % y
    return x
```

```python
def crack_unknown_multiplier(states, modulus):
    multiplier = (states[2] - states[1]) * modinv(states[1] - states[0], modulus) % modulus
    increment = crack_unknown_increment(states, modulus, multiplier)
    return increment, multiplier
```

LCG PRNG cracking

```
def crack_unknown_modulus(states):
    diffs = [s1 - s0 for s0, s1 in zip(states, states[1:])]
    zeroes = [t2*t0 - t1*t1 for t0, t1, t2 in zip(diffs, diffs[1:], diffs[2:])]
    modulus = abs(functools.reduce(gcd, zeroes))
    increment, multiplier = crack_unknown_multiplier(states, modulus)
    return modulus, increment, multiplier
```

```
[x] Cracking LCG PRNG
[x] Setting the inputs - random number)
[x] Recovered modulus    :  8512677386048191063
[x] Recovered increment  :  1035085024576065627
[x] Recovered multiplier :  317069504227672257
```

Mersenne Twister

Mersenne Twister I

- The Mersenne Twister is a pseudorandom number generator (PRNG)
- It has a very long period $2^{19937} - 1$
- **k-distributed** to 32-bit accuracy for every $1 \leq k \leq 623$
- Really fast
- Python random.Random() is based on MT.

Mersenne Twister II

- Excellent statistical properties (indistinguishable from truly random)
- However, it is NOT CPRNG.

Steps to Crack MT:

a) **randcrack** package to crack it in python!

b) feed cracker with 624 randomly generated numbers (32-bit integers)

c) After submitting 624 integers it will be ready for predicting new numbers.

Mersenne Twister Cracker

```
from randcrack import RandCrack

random.seed(100)
rc = RandCrack()

print("[x] feeding the cracker with 624 randomly generated numbers")
for i in range(624):
    r = random.getrandbits(32)
    rc.submit(r)

print("[x] Now, it is ready to predict new randomly generated numbers ..")
print("\n[x] Random Number: {}\n[x] Predicted number: {}"
    .format(random.randrange(0, 4294967295), rc.predict_randrange(0, 4294967295)))
```