Chalk & Talk

Linear Congruential Generators

CS409M - Introduction to Cryptography

September, 2024

Shubhhi Singh (210070084)
Shravya Suresh (210260046)

Contents

1. Introduction

A Linear Congruential Generator (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers using the recurrence relation:

$$X_{n+1} = (aX_n + b) \mod m$$

where:

- m: Modulus
- a: Multiplier
- b: Increment
- X_0: Seed

1.1 Characteristics of LCGs

- **Fast:** LCGs are computationally efficient and easy to implement.
- **Widely Available:** LCGs are commonly used in applications that do not require strong notions of security.
- **Not Cryptographically Secure:** LCGs are predictable, making them unsuitable for cryptographic applications.

1.2 Defining the LCG

The LCG is defined by four parameters:

- **Modulus** (m): A large positive integer.
- **Multiplier** (a): An integer in the range $0 < a < m$.
- **Increment** (b): An integer in the range $0 \leq b < m$.
- **Seed** (X_0): The initial value from which the sequence starts.

  The recurrence relation for generating the sequence is:

$$X_{n+1} = (aX_n + b) \mod m$$

2. Cryptographic Insecurity of LCGs

LCGs are not suitable for cryptographic applications because they are inherently insecure as pseudo-random number generators (PRNGs). Their predictability makes them vulnerable to attacks, particularly when an attacker can gain knowledge of the internal parameters or assumptions about the output.

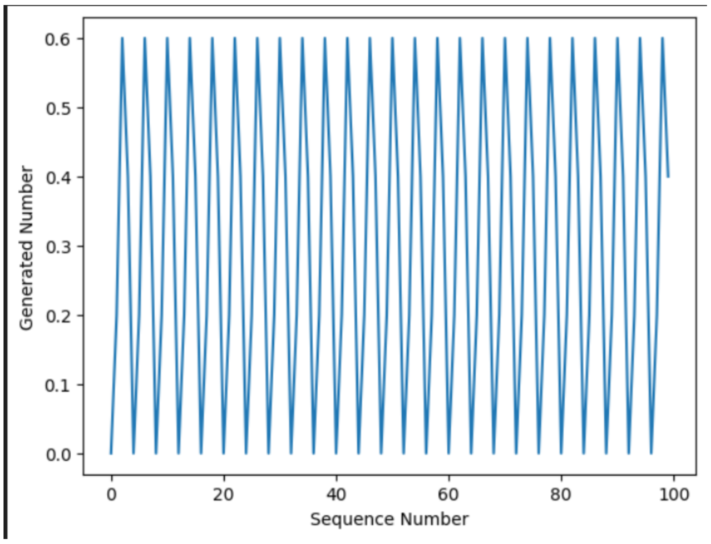

Figure 1: LCG output with parameters $X_0 = 4$, $a = 2$, $b = 2$, $m = 10$

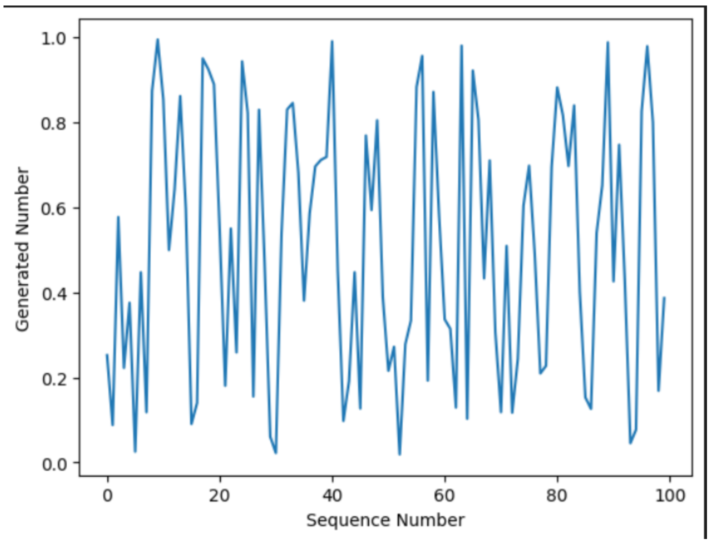

Figure 2: LCG output with parameters $X_0 = 42$, $a = 1664525$, $b = 1013904223$, $m = 2^{32}$

3. Lattice-Based Attack

3.1 Assumptions

Consider an LCG where:

- The output is the higher-order bits of the general LCG sequence.
- The modulus m is large.
- The parameters w, a, b are known, with $w \leq \lfloor \sqrt{m}/c \rfloor$ for some fixed c.

3.2 General Idea of Breaking the Sequence

The general idea of breaking an LCG sequence is to predict the next value using two consecutive outputs, r_i and r_{i+1}:

$$r_i = \left\lfloor \frac{x_i}{w} \right\rfloor, \quad r_{i+1} = \left\lfloor \frac{(ax_i + b) \mod m}{w} \right\rfloor$$

For some unknown x_i
We can derive relationships to estimate the value of the parameters and thereby predict the remainder of the sequence.

Using two consecutive outputs, the following relationships hold:

$$r_i w + e_0 = x_i, \quad r_{i+1} w + e_1 = (ax_i + b) \mod m$$

where $0 \leq e_0, e_1 \leq w$.
Let us write x in place of x_i, and eliminate the $\mod m$ by introducing an integer variable k, to obtain

$$r_i \cdot w + e_0 = x$$

and

$$r_{i+1} \cdot w + e_1 = ax + b + mk.$$

The integers x, k, e_0, e_1 are unknown to the attacker, but they have the knowledge of the integers r_i, r_{i+1}, w, a, b. Rearranging terms, we get:

$$x = r_i \cdot w + e_0$$

and

$$ax + mk = r_{i+1} \cdot w - b + e_1.$$

Finally, we can write in vector form as

$$\begin{bmatrix} r_i w \\ r_{i+1} w - b \end{bmatrix} = x \cdot \begin{bmatrix} 1 \\ a \end{bmatrix} + k \cdot \begin{bmatrix} 0 \\ m \end{bmatrix} = \mathbf{g} + \mathbf{e}$$

where,

$$\mathbf{g} := \begin{bmatrix} r_i w \\ r_{i+1} w - b \end{bmatrix}, \quad \mathbf{g} := \begin{bmatrix} e_0 \\ e_1 \end{bmatrix}$$

3.3 Approach Using Lattices

The attacker knows $\mathbf{g} \in \mathbb{Z}^2$, but it does not know sk, x, or $\mathbf{e} \in \mathbb{Z}^2$. However, it knows that $\mathbf{e}$ is short, i.e $\|\mathbf{e}\|_\infty < \sqrt{m}/c$.

Let $\mathbf{u} \in \mathbb{Z}^2$ denote the unknown vector $\mathbf{u} := \mathbf{g} + \mathbf{e} = x \cdot (1, a)^T + k \cdot (0, q)^T$. If the attacker could find $\mathbf{u}$, then it could easily recover k and x from $\mathbf{u}$. Given x, it could predict the rest of the PRG output. Thus, to break the generator it suffices to find the vector $\mathbf{u} \in \mathbb{Z}^2$. The attacker has $\mathbf{g} \in \mathbb{Z}^2$, and it knows that $\|\mathbf{g} - \mathbf{u}\|_\infty = \|\mathbf{e}\|_\infty$ is short, meaning that $\mathbf{u}$ is "close" to $\mathbf{g}$.

The problem can be reduced to finding the closest vector in a lattice $\mathcal{L}_a$ generated by the vectors $(0, m)^T$ and $(1, a)^T$. The attacker needs to find $\mathbf{u}$ closest to $\mathbf{g}$ in $\mathcal{L}_a$.

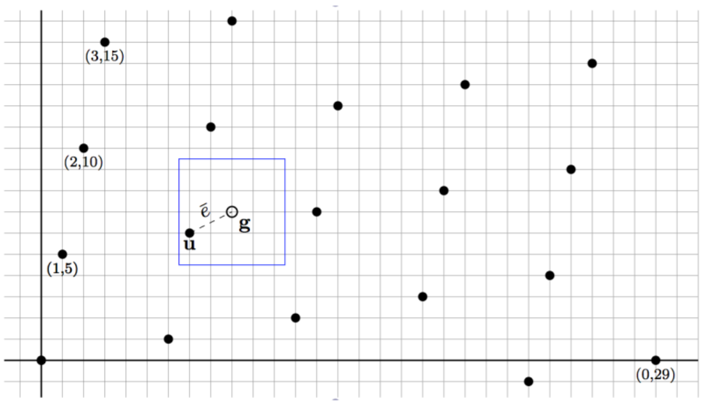

Figure 3: 2-D lattice $\mathcal{L}_a$ generated by $(1,5)^\top$ and $(0,29)^\top$, the points correspond to integral linear combinations of the generating vectors

3.4 Finding the Closest Vector

To break the LCG sequence, the attacker must find the vector u closest to g in the lattice $\mathcal{L}_a$. If the attacker can find u, they can easily recover the values of k and x.

We have the following result:
Lemma: For at least $(1 - 16/c^2) \cdot m$ of the a in S_q, the lattice $\mathcal{L}_a \subset \mathbb{Z}^2$ has the following property: for every $\mathbf{g} \in \mathbb{Z}^2$, there is at most one vector $\mathbf{u} \in \mathcal{L}_a$ such that $\|\mathbf{g} - \mathbf{u}\|_\infty < \lfloor \sqrt{m}/c \rfloor$.

For completeness, $a = 1$ and $a = 2$ are examples where Lemma fails. For these values, there may be multiple lattice vectors in $\mathcal{L}_a$ close to a given $\mathbf{g} \in \mathbb{Z}^2$, and the attack may fail.
Proof : Let $\mathbf{g} \in \mathbb{Z}^2$ and suppose there are two vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ in $\mathcal{L}_a$ close to $\mathbf{g}$, i.e., $\|\mathbf{u}_i - \mathbf{g}\|_\infty < \lfloor \sqrt{m}/c \rfloor$ for $i = 0, 1$. By the triangle inequality:

$$\|\mathbf{u}_0 - \mathbf{u}_1\|_\infty \leq \|\mathbf{u}_0 - \mathbf{g}\|_\infty + \|\mathbf{g} - \mathbf{u}_1\|_\infty < 2\lfloor \sqrt{m}/c \rfloor.$$

Since $\mathcal{L}_a$ is closed under addition, $\mathbf{u} := \mathbf{u}_0 - \mathbf{u}_1$ is a non-zero vector in $\mathcal{L}_a$ with norm less than $B := 2\lfloor \sqrt{m}/c \rfloor$.

Considering the case when m is a prime. We show that every short vector in $\mathbb{Z}^2$ is contained in at most one lattice $\mathcal{L}_a$. Therefore, the number of bad a's is at most the number of short vectors in $\mathbb{Z}^2$.

1. Let $\mathbf{t} = (s, y)^\top \in \mathbb{Z}^2$ be some non-zero vector such that $\|\mathbf{t}\|_\infty < B$.
2. Suppose that $\mathbf{t} \in \mathcal{L}_a$ for some $a \in S_m$.
3. Then there exist integers x_a and k_a such that

$$x_a \cdot (1, a)^\top + k_a \cdot (0, q)^\top = \mathbf{t} = (s, y)^\top.$$

4. From this, we obtain that $s = x_a$ and $y = as \mod q$.
5. Moreover, $s \neq 0$ since otherwise $\mathbf{t} = 0$.

6. Since $y = as \mod q$ and $s \neq 0$, the value of a is uniquely determined, namely,
$$a = ys^{-1} \mod q.$$
7. Hence, when m is prime, every non-zero short vector $\approx$ is contained in at most one lattice $\mathcal{L}_a$ for some $a \in S_m$.
8. It follows that the number of bad a in S_m is at most the number of vectors $\mathbf{t} \in \mathbb{Z}^2$ where $\|\mathbf{t}\|_\infty < B$, which is at most $(2B)^2 \leq \frac{16q}{c^2}$.

   The same bound on the number of bad a's holds when m is not a prime.
9. Consider a specific non-zero $s \in S_m$ and let $d = \gcd(s, m)$.
10. As above, a vector $\mathbf{t} = (s, y)^\top$ is contained in some lattice $\mathcal{L}_a$ only if there is an $a \in S_m$ satisfying
$$as \equiv y \pmod{q}.$$
11. This implies that y must be a multiple of d, so that we need only consider $\frac{2B}{d}$ possible values of y.
12. For each such y, the vector $\mathbf{t} = (s, y)^\top$ is in at most d lattices L_a.
13. Since $\|\mathbf{t}\|_\infty < B$, there are at most $2B$ possible values for s.
14. Hence, the number of bad a's is bounded by
$$(d \cdot \frac{2B}{d}) \cdot 2B = (2B)^2,$$
as in the case when q is prime.

Thus, there are at most $\frac{16m}{c^2}$ bad values of a in S_m. Therefore, for at least $(1 - \frac{16}{c^2}) \cdot m$ of the a values in S_m, the lattice $\mathcal{L}_a$ contains no non-zero short vectors, and the lemma follows.

Because of the lemma, it remains to efficiently find the closest vector to $\mathbf{g}$ in $\mathcal{L}_a$. This is an instance of the closest vector problem i.e given a lattice $\mathcal{L}$ and a vector $\mathbf{g}$, find the vector in $\mathcal{L}$ closest to $\mathbf{g}$. In two dimensions, there is an efficient polynomial-time algorithm for this problem. Thus, the problem reduces to finding the closest vector to g in $\mathcal{L}_a$ and recovering x of the LCG generator using two outputs r_i and r_{i+1}.

4. Generalising the Attack

In this section, we highlight examples of attacks on an LCG under various possible conditions of known and unknown parameters.

The aim of the adversary is to be able to predict the rest of the sequence given a certain set of initial elements of the sequence $\{X_n\}$. This is equivalent to using an initial segment of the sequence to find values a_c, b_c such that $\forall n \geq 0$:
$$X_{n+1} = a_c X_n + b_c \bmod m$$

Note that this equation does not require us to know the value of m. The algorithm can further be modified to predict both an unknown m and the unknown elements of the sequence.

4.1 When a, b, m are known

This is a trivial case, for if all parameters are known, the adversary can easily generate the next element of the sequence, and the system hence becomes deterministic.

4.2 When b is unknown, a, m are known

When b is unknown, we can find its value by generating just the first two elements of the sequence $\{X_n\}$.

$$X_1 = a_c X_0 + b_c \bmod m$$

which on rearranging gives

$$b_c = X_1 - a_c X_0 \bmod m$$

Thus, we now know all three parameters of the LCG, and can hence predict the rest of the sequence.

4.3 When a, b are unknown, m is known

When m is known, we set out to determine values a_c and b_c that can give us the remaining elements of the sequence. We define a sequence $\{Y_n\}$ where $\forall n \geq 0$:

$$Y_{n+1} = X_{n+1} - X_n$$

which gives us $\forall k \geq 1$:

$$Y_{k+1} = a_c Y_k \text{ mod m}$$

We thus need just the first two elements of sequence $\{Y_k\}$, and hence the first three element of sequence $\{X_n\}$ to find a_c.

$$X_1 = a_c X_0 + b_c \bmod m$$

$$X_2 = a_c X_1 + b_c \bmod m$$

Subracting these equations,

$$X_2 - X_1 = a_c(X_1 - X_0) \bmod m$$

which on rearranging gives

$$a_c = \tfrac{X_2 - X_1}{X_1 - X_0} \bmod m$$

Having found a_c, we can then find b_c as described in the previous section, and hence we can break the LCG.

4.4 When a, b, m are all unknown

This is the tricky part. On first sight, we'd think that 3 unknown means we need 3 equations to solve them, so we just need to generate the first four elements of the sequence. However, if we reorder the equation of the LCG, we get:

$$X_1 - (a_c X_0 + b_c) = k_1 m$$
$$X_2 - (a_c X_1 + b_c) = k_2 m$$
$$X_3 - (a_c X_2 + b_c) = k_3 m$$

meaning 3 equations but 6 unknowns. So we take a different approach which involves a little number theory. Before getting to the main point, let's set up some foundations:

Consider any two natural numbers p, q which are multiples of m. Then, the probability that the GCD of p and q is m is deterministic:

$$P(\text{GCD}(p,q) = m) = \tfrac{6}{\pi^2}$$

Proof: Let us rewrite p and q as

$$p = n_1 \times m$$
$$q = n_2 \times m$$

Now, our proof reduces to finding the probability of $n1$ and $n2$ being coprime:

$$P(\text{GCD}(n_1, n_2) = 1) = \tfrac{6}{\pi^2}$$

Let us now prove this. Observe the set of natural numbers. We notice that the probability of any number being divisible by 1 is 1, by 2 is 1/2 (all even numbers only, which is half the set of natural numbers), and so on. Thus, we can generalise this to say:

$$P(\text{A Natural number is divisible by p}) = \tfrac{1}{p}$$

Now, by independence, the probability that two numbers would be divisible by p would be the product of their respective probabilities of being divisible by p:

$$P(\text{Two Natural numbers are divisible by p}) = \tfrac{1}{p^2}$$

Thus, the probability that two natural numbers n_1, n_2 are both not divisible by p is

$$P(n_1, n_2 \text{ are not divisible by } p) = 1 \text{ - } \tfrac{1}{p^2}$$

In order for n_1 and n_2 to be coprime, they should both not be divisible by all numbers, which can be prime factorised such that we consider only prime numbers. Again, by independence of probability, this result would be equal to the product of the above probability over all prime numbers.

$$P(\text{GCD}(n_1, n_2) = 1) = \prod_{p, prime} (1 - \tfrac{1}{p^2})$$

which can be rewritten as

$$P(\text{GCD}(n_1, n_2) = 1) = (\prod_{p, prime} (\tfrac{1}{1-p^{-2}}))^{-1}$$

By Euler's identity:

$$\prod_{p,prime} (\tfrac{1}{1-p^{-2}}) = \sum_{n=1}^{\infty} \tfrac{1}{n^2} = \zeta(2) = \tfrac{\pi^2}{6}$$

Hence,

$$(\prod_{p,prime} (\tfrac{1}{1-p^{-2}}))^{-1} = \tfrac{6}{\pi^2}$$

Thus, we have proved that

$$P(\text{GCD}(n_1, n_2) = 1) = \tfrac{6}{\pi^2}$$

implying that

$$P(\text{GCD}(p, q) = m) = \tfrac{6}{\pi^2}$$

For more multiples of m, the probability of all of their GCD being m increases until it becomes very close to 1. So, a finite number of multiples will be sufficient for our use case.

$Y_0 = X_1 - X_0$
$Y_1 = X_2 - X_1 = (aX_1 + b) - (aX_0 + b) = a * (X_1 - X_0) = aY_0 \text{ (mod m)}$
$Y_2 = X_3 - X_2 = (aX_2 + b) - (aX_1 + b) = aY_1 \text{ (mod m)} = a^2Y_0 \text{ (mod m)}$
$Y_3 = X_4 - X_3 = (aX_3 + b) - (aX_2 + b) = aY_2 \text{ (mod m)} = a^3Y_0 \text{ (mod m)}$

Thus,

$$Y_2 \times Y_0 - Y_1 \times Y_1 = (a^2Y_0 \times Y_0) - (aY_0 \times aY_0) = 0 \text{ (mod m)}$$

Similarly, the above equation holds for $\{Y_1, Y_2, Y_3\}$, $\{Y_2, Y_3, Y_4\}$ and so on. Thus, as we increase the number of samples we use in the above equation, we converge to a suitable value m_c that can help us find suitable corresponding values of a_c and b_c in the methods described previously. We performed the above calculation using the first seven elements of the sequence $\{X_n\}$, meaning the triplets $\{Y_0, Y_1, Y_2\}$, $\{Y_2, Y_3, Y_4\}$, $\{Y_2, Y_3, Y_4\}$, $\{Y_3, Y_4, Y_5\}$ and $\{Y_4, Y_5, Y_6\}$ to converge to a suitable value of m_c. The code is included in the appendix.

One important thing to note here is that the values of a_c, b_c and m_c need not even be the exact solution of the LCG. These just need to be values that help us find all the remaining elements of the sequence.

5. Conclusion

Through this report, we have investigated possible attacks upon a Linear Congruential Generator (LCG) through two different approaches, and under various circumstances of known and unknown parameters. Hence, one is strongly advised against using LCGs in their security systems for the purpose of encryption.

6. Acknowledgements

We are indebted to Prof. Sruthi Sekar for giving us the opportunity to delve into LCGs and their insecurity through this Chalk and Talk. We enjoyed the process of learning how to attack LCGs, as well as working together on a presentation.

7. References

[1] Joan B. Plumstead "Inferring a sequence generated by a Linear Congruence"

[2] Dan Boneh and Victor Shoup "A graduate course in applied cryptography"

[3] James Reeds "Cracking a Random Number Generator"

[4] Donald E. Knuth "Deciphering a Linear Congruential Encryption"

[5] Cracking RNGs: Linear Congruential Generators

8. Appendix

Code for Generating LCG Output

```python
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt

def lcg(seed, a, c, m, n):
    numbers = []
    current = seed
    for _ in range(n):
        current = (a * current + c) % m
        r_number = current / m
        numbers.append(r_number)
    return numbers

# Defining the parameters for the LCG
seed = 4
a = 2
c = 2
m = 10
n = 100  # Number of random numbers to generate

generated_numbers = lcg(seed, a, c, m, n)
print("Generated numbers with poor parameters:", generated_numbers)

# Defining the parameters for the LCG
seed = 42
a = 1664525
c = 1013904223
m = 2**32
```

```
n = 100  # Number of random numbers to generate

generated_numbers_better = lcg(seed, a, c, m, n)
print("Generated numbers with better parameters:",
    generated_numbers_better)
```

Code for generalised attacks on the LCG

```
import numpy as np
import functools as f
import sys
from typing import Tuple
sys.setrecursionlimit(1000000)

# Expected values
X0 = 2300417199649672133
X1 = 2071270403368304644
X2 = 5907618127072939765

## When all parameters - m, a, b - are known

a = 672257317069504227  # the multiplier
b = 7382843889490547368  # the increment
m = 9223372036854775783  # the modulus
X0 = 2300417199649672133 # seed

X1 = (X0*a + b) % m
X2 = (X1*a + b) % m
X3 = (X2*a + b) % m
X4 = (X3*a + b) % m


## When Increment - b - is unknown

a = 81853448938945944
# b = unknown
m = 9223372036854775783
# Given values
X0 = 4501678582054734753
X1 = 4371244338968431602

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

print(find_unknown_increment([4501678582054734753,
    4371244338968431602], 9223372036854775783, 81853448938945944))


## When increment - b - and multiplier - a - are unknown

m = 9223372036854775783

# Known values
X0 = 6473702802409947663
X1 = 6562621845583276653
```

```python
X2 = 4483807506768649573

def mygcd(a: int, b: int) -> Tuple[int, int, int]:
    if a == 0:
        return (b, 0, 1)
    else:
        b_div_a, b_mod_a = divmod(b, a)
        g, x, y = mygcd(b_mod_a, a)
        return (g, y - b_div_a * x, x)

def modinv(a: int, b: int) -> int:
    g, x, _ = mygcd(a, b)
    if g != 1:
        raise Exception('gcd(a, b) != 1')
    return x % b

def find_unknown_multiplier(states, modulus):
    multiplier = (states[2] - states[1]) * modinv(states[1] - states[0], modulus) % modulus
    return find_unknown_increment(states, modulus, multiplier)

print (find_unknown_multiplier([6473702802409947663, 6562621845583276653, 4483807506768649573], 9223372036854775783))


## When a, b, m all are unknown

# Known values
X0 = 2818206783446335158
X1 = 3026581076925130250
X2 = 136214319011561377
X3 = 359019108775045580
X4 = 2386075359657550866
X5 = 1705259547463444505
X6 = 2102452637059633432

def find_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(f.reduce(np.gcd, zeroes))
    return find_unknown_multiplier(states, modulus)

print (find_unknown_modulus([2818206783446335158, 3026581076925130250,
    136214319011561377, 359019108775045580, 2386075359657550866,
    1705259547463444505]))
```