Backtracking Approach for Seed Recovery in a Pseudo-Random Number Generator with Hash-Based Verification

Anindya Naufal Pinasthika - 13524013
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung
E-mail: 13524013@mahasiswa.itb.ac.id, naufalpinasthika@gmail.com

***Abstract*—A Pseudo-Random Number Generator (PRNG) is an algorithm designed to simulate true randomness through mathematical formulas, producing sequences of numbers that appear random. However, due to the fundamentally deterministic nature of computing systems, true randomness is unattainable via these methods. PRNG relies on an initial value known as PRNG's seed. Because the process is deterministic, the output sequence is entirely predictable if the seed is discovered. This characteristic introduces significant security concerns, particularly in contexts where PRNG are utilized as verification mechanisms for hash functions, cryptographic operations that are intended to be irreversible. If the PRNG system is not secure enough (i.e. flawed design by introducing partial constraint or the bit generation is not random enough) then security concerns would arise as the PRNG could be retrieved. One of the ways a PRNG seed could be retrieved is by an algorithm known as backtracking. By utilizing a backtracking algorithm, a PRNG's seed could be retrieved by taking advantage of its constraint that PRNG used as an encryption.**

***Keywords*—Psuedo-Random Number Generator, Hash, Cryptography, Backtracking**

I. Introduction

Pseudo-Random Number Generator (PRNG) is an algorithm (usually deterministic) which tries to emulate the statistical properties of a sequence of True-Random Numbers (TRNs). More specifically, Pseudo-Random Number is one of the most foundational fields in computer science and cryptography used in applications related to information security, statistical sampling, computer simulation and in other areas where producing an unpredictable result is desirable. While TRN sequences are overall more unpredictable and as such better keys for cryptography systems, they are usually expensive to generate. A True Random Number Generator (TRNG) relies on natural phenomena like atmospheric or thermal noise, radioactive decay or cosmic background radiation. Measurement of these is known to be expensive. As a consequence TRNGs are employed only in applications of vital importance while PRNGs are used in almost every other setting [1].

A Deterministic Random Bit Generator (DRBG) is a type of Pseudo-Random Number Generator (PRNG) that utilizes a defined DRBG algorithm to produce a sequence of random bits. Likewise, a core property of a DRBG is its initial input value, known as the DRBG seed [2].

One common use case for a DRBG (and PRNGs in general) is hash-based function verification, where the intention is to produce secure, one-way information that is computationally infeasible to reverse or break [3].

While this is effective in theory, a poorly designed DRBG system can lead to severe cryptographic security issues. If the entropy of the hash function or environment is insufficient, the DRBG seed can be retrieved using backtracking techniques, making seed recovery possible without requiring significant time or computational power. This is because of the nature of backtracking attacks, which can quickly eliminate incorrect paths or expand valid potential seed bits. As long as a candidate bit segment meets the verification requirements needed to match the generated PRNs, the algorithm can rapidly narrow down and recover the actual seed.

This paper focuses on implementing a Pseudo-Random Number Generator (PRNG) based on the Linear Congruential Generator (LCG) that will purposely have its poorly designed. The LCG is selected because its recurrence relation is simple, deterministic, and suitable for algorithmic analysis. In real cryptographic systems, standardized Deterministic Random Bit Generator (DRBG) constructions, such as Hash_DRBG, HMAC_DRBG, or CTR_DRBG, are more appropriate for secure random bit generation [3]. However, this paper intentionally designs a DRBG-inspired system in which an LCG is used as the internal deterministic generator, while a hash-based verification layer is placed on the outside.

This design makes the generated output appear random and cryptographically structured because the system includes seed-based generation, internal state updates, and a hash verification. However, the security weakness remains in the LCG-based internal generator. Since the generator is deterministic and its seed space is limited, known plaintext constraints can be used to prune invalid

seed candidates. Therefore, this paper demonstrates how backtracking can recover the seed in a controlled cryptographic challenge environment, while hash functions are only used to validate the final recovered seed and plaintext.

II. Fundamental Theorem

A. Linear Congruential Generator (LCG)

A Linear Congruential Generator (LCG) is one of the simplest methods used to generate pseudo-random numbers. The generator produces a sequence of integer values x_n, which can be converted into real values in the interval $[0, 1)$ by computing $U_n = \frac{x_n}{m}$

$$x_{n+1} = (ax_n + c) \bmod m$$

With $n \geq 0$ and the values defined as:
x_0: initial value or seed,
x_n : generated value at certain n,
a : Multiplier,
c : Increment,
m : Modulus

Because the state x_n is always limited to the range 0 to $m - 1$, the sequence must eventually repeat. Once a value appears again, the generator enters a cycle. Therefore, the period of an LCG is at most m. A longer period is usually preferred because it allows the generated sequence to appear more random.

For example, the sequence obtained when $x_0 = a = c = 7, m = 10$, is:

7, 6, 9, 0, 7, 6, 9, 0, …

The quality of an LCG is strongly influenced by the selected parameters. Although a long period is desirable, it does not automatically imply that the generated sequence has good randomness properties. Therefore, additional factors such as the multiplier, increment, modulus, and structural properties of the sequence must also be considered.

The randomness quality of a generated sequence can be evaluated using empirical tests. Common examples include the frequency test, serial test, gap test, poker test, and chi-square test. These tests examine whether the generated numbers are distributed in a way that resembles random behavior [4]. Nevertheless, a good quality of randomness produced LCG will be hard to notice its pattern when using scatter plot or correlogram as shown in Fig 2.1 and Fig 2.2.

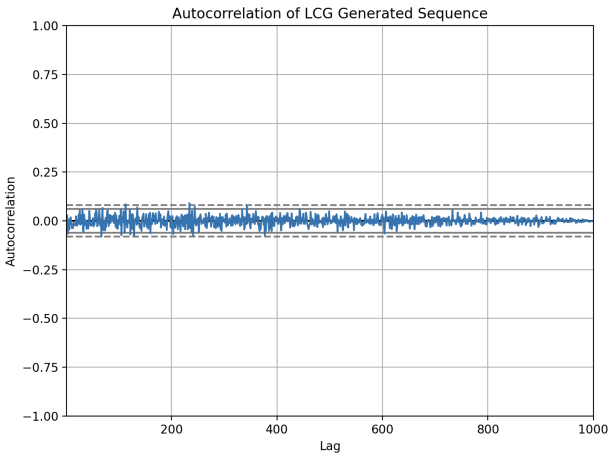

Fig 2.1 : Correlogram/Autocorrelation representation of generated LCG
Source : QuantStart. (2016). Linear Congruential Generators in Python. QuantStart Articles. Available: https://www.quantstart.com/articles/linear-congruential-generators-in-python/

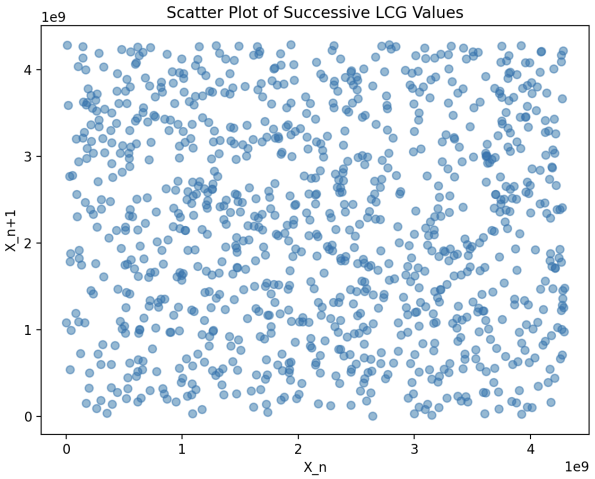

Fig 2.2 : Scatter plot representation of generated LCG
Source : QuantStart. (2016). Linear Congruential Generators in Python. QuantStart Articles. Available: https://www.quantstart.com/articles/linear-congruential-generators-in-python/

B. Hash Function

A cryptographic hash function is a deterministic function that maps an input message of arbitrary length into a fixed-length output called a hash value or message digest. Even a small change in the input should produce a significantly different digest, making hash functions useful for verifying data integrity.

In cryptographic applications, hash functions are commonly used to detect message modification, build digital signatures, construct message authentication schemes, and verify whether a given input matches a stored digest [5].

A secure cryptographic hash function is usually expected to satisfy several security properties, including preimage resistance, second-preimage resistance, and collision resistance.

Preimage resistance states that given a hash value h, it

should be computationally infeasible to find an input x, such that

$$H(x) = h$$

which means a hash function will be irreversible.

The second property is second-preimage resistance, which means that given an input x, it should be computationally infeasible to find another input y, where

$$x \neq y,$$

such that

$$H(x) = H(y)$$

The third property is collision resistance, which means that it should be computationally infeasible to find any two different inputs x and y that produce the same hash value [6].

C. Backtracking

Backtracking is an algorithmic technique used to solve problems by exploring possible solutions incrementally. In general, a backtracking algorithm has three main properties

1) *Solution Representation*

The solution is represented as an n-tuple vector:

$$X = (x_1, x_2, \ldots, x_n)$$

where each component x_i is selected from a candidate set S_i. In many problems, the candidate sets are the same for every component for

$$S_1 = S_2 = \ldots = S_n$$

2) *Value Generation Function*

The value generation function is used to generate possible values for the next component of the solution vector. This function can be denoted as:

$$T(x_1, x_2, \ldots, x_{k-1})$$

The function generates possible values for x_k, which is the next component to be added to the partial solution.

3) *Bounding Function*

The bounding function, or constraint function, is used to determine whether a partial solution can still lead to a valid complete solution. It can be written as a predicate:

$$T(x_1, x_2, \ldots, x_k)$$

The function returns true if the partial solution

$$(x_1, x_2, \ldots, x_k)$$

still satisfies the problem constraints. If the function returns true, the search continues by generating the next component x_{k+1}. If the function returns false, the partial solution is discarded because it cannot lead to a valid solution.

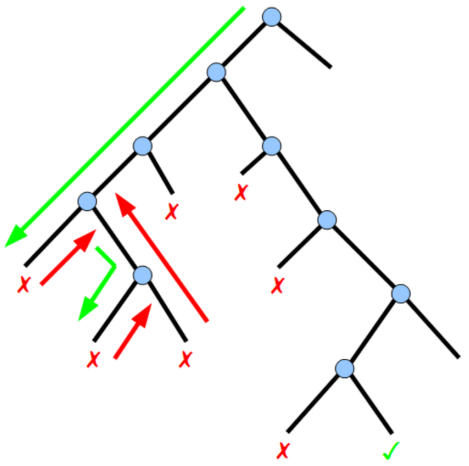

Fig 2.2 : Illustration of backtracking algorithm

Source : Pemberton, S. (2011). The Computer as Extended Phenotype (Computers, Genes and You). World Wide Web Consortium (W3C) Talks. Available: https://www.w3.org/2011/Talks/01-14-steven-phenotype/

The search process can be represented as a state space tree, where each node represents a partial solution and each path from the root to a leaf represents one possible complete solution . Backtracking explores this state space tree in a depth-first manner. At each step, the algorithm extends the current partial solution by selecting a possible candidate for the next component. After a candidate is added, the algorithm checks whether the partial solution still satisfies the given constraints. If the partial solution violates a constraint or cannot lead to a valid complete solution, the algorithm stops exploring that branch and returns to the previous state. This process is called backtracking [7], [8].

Below is a pseudo code demonstrating the backtracking algorithm.

```
procedure Backtrack(partial_candidate):
    if partial_candidate violates
constraints:
        return

    if partial_candidate is complete:
        plaintext = decrypt(ciphertext,
partial_candidate)

        if hash(partial_candidate ||
plaintext) == target_hash:
            output partial_candidate and
plaintext

        return

    for each possible next value:
        Backtrack(partial_candidate +
next value)
```

In general, the worst-case complexity of backtracking

can remain exponential for combinatorial search problems. However, effective pruning can significantly reduce the actual number of explored states. This makes backtracking suitable for problems where a solution must satisfy a set of constraints, such as N-Queens, subset sum, maze solving, and constrained seed recovery [8], [9].

III. Implementation

A. Initial Design and Tools

We will be using Python as the main programming language for simplicity and readability. Python also gives a useful number of cryptographic libraries (such as hashlib and xor) that will be important when processing LCG.

```
import hashlib
Import secrets
import string
from functools import reduce
from operator import xor
```

When constructing the LCG, we will be using Knuth and H.W. Lewis test where

```
a = 1664525
c = 1013904223
m = 2**32
```

Will be about as good as any 32-bit linear congruential generator and will be efficiently fast [10].

B. LCG Architecture

The architecture is designed as a classic stream cipher that incorporates a progressive seed injection mechanism. This is because pure LCG is fundamentally unsuitable for cryptographic purposes due to LCG having a rigid algebraic linearity [11]. Instead of utilizing a static seed to expand a monolithic keystream, the system ingests the secret seed vector character-by-character.

Let X_i represent the internal state of the generator at step i, initialized by a public Initialization Vector

$$X_0 = IV.$$

For each character s_i in the secret seed vector at index i, the next state X_{i+1} is computed via a pure linear congruential recurrence relation:

$$X_{i+1} = (a \cdot X_i + c + ord(s_i)) \,\&\, m$$

To generate the keystream byte k_i for encrypting the plaintext, the system isolates the most significant 8 bits (the high byte) of the newly computed state:

$$k_i = (X_{i+1} >> 24) \,\&\, 0xFF$$

The ciphertext byte c_i is then produced by performing an exclusive-or (XOR) operation between the plaintext character p_i and the keystream byte k_i:

$$c_i = p_i \oplus k_i$$

In the end of the encryption, the product of the LCG encryption then will be verified using hashlib.sha256 function. Below is the snippet of the python program to demonstrate the encryption process:

```
A = 1664525
C = 1013904223
M = (2**32) - 1 # bitwise mask
IV = 1821293813

state = IV
ciphertext = bytearray()

for i in range(SEED_LENGTH):
    state = (A * state + C +
ord(SECRET_SEED[i])) & M

    k_i = (state >> 24) & 0xFF

    ciphertext.append(ord(PLAINTEXT[i]) ^
k_i)

target_hash =
hashlib.sha256(f"{SECRET_SEED}|{PLAINTEXT
}".encode()).hexdigest()
```

With SECRET_SEED and PLAINTEXT can be initialized beforehand accordingly.

C. Backtracking Feasibility

Backtracking method operates will feasible when there is set of "heuristic" applied. That is, when there is structural constraints or known plaintext format rules are available during the execution. In a poorly designed cryptographic system, an attacker leveraging these known constraints can systematically identify and retrieve the hidden seed vector, which subsequently enables them to fully decipher the ciphertext.

Because the keystream byte k_i at depth i is generated deterministically from the single seed character s_i and the previous state X_i, the algorithm does not need to guess the full 14-character seed globally. Instead, it evaluates each candidate component locally. If a guessed seed character yields a predicted plaintext byte that violates the established formatting rules (the bounding function), the entire sub-tree rooted at that invalid choice is instantly pruned. This localized bounding function collapses the exponential brute-force search space into a highly practical execution timeline. Once the complete seed vector is successfully recovered, the attacker can perfectly reproduce the entire cryptographic keystream, thereby fully deciphering the target plaintext.

Backtracking algorithm used to retrieve a set of seed will result as follows:

```
def backtrack(position, state,
seed_prefix, plaintext_prefix):
    global nodes_explored
    global branches_pruned
    global complete_candidates

    nodes_explored += 1

    if position == len(CIPHERTEXT):
        complete_candidates += 1

        seed = "".join(seed_prefix)
        plaintext =
"".join(plaintext_prefix)

        candidate =
f"{seed}|{plaintext}".encode("ascii")
        candidate_hash =
hashlib.sha256(candidate).hexdigest()

        if candidate_hash == TARGET_HASH:
            return seed, plaintext

        return None

    for seed_char in string.digits:
        next_state = (A * state+ C +
ord(seed_char) + position) & MASK

        key_byte = ( next_state^
(next_state >> 8)^ (next_state >> 16)^
(next_state >> 24)) & 0xFF

        plain_byte = CIPHERTEXT[position]
^ key_byte

        if not 32 <= plain_byte <= 126:
            branches_pruned += 1
            continue

        plain_char = chr(plain_byte)

        if not
valid_plaintext_character(position,
plain_char):
            branches_pruned += 1
            continue

        seed_prefix.append(seed_char)
```

```
plaintext_prefix.append(plain_char)

        result = backtrack(
            position + 1,
            next_state,
            seed_prefix,
            plaintext_prefix
        )

        if result is not None:
            return result

        seed_prefix.pop()
        plaintext_prefix.pop()

    return None
```

IV. Result Analysis

A. Experimental Setup

LCG generator will automatically produce a random decimal seed with the same length as the plaintext. One generated instance used will produce the following:

```
=== PUBLIC DATA ===
CIPHERTEXT_HEX =
'180c6d39ac8c38bc50c6dad84d1097a3e1538ad0
365a94d95fcd04fe46d5250e8284'
TARGET_HASH    =
'd83205376f1266d56fb2af9ba5e4139f85cf5c59
24093d63b127b5817abbb35b'
====================
```

B. Backtracking result

After extracting a certain CIPHERTEXT_HEX and TARGET_HASH, we will be using a heuristic that plaintext have a format output by “STIMA{“. The result will produce the following:

```
    Recovered Seed      :
998561898127161029633118351340011096O
    Deciphered Plaintext:
STIMA{th1s_1s_4_53cr3t_m355493555sss}
```

With the messages that have been encrypted and leaked via recovering the state seed of an LCG are:

```
STIMA{th1s_1s_4_53cr3t_m355493555sss}
```

V. Conclusion

Backtracking is one of many ways of algorithm that can be used to recover a hidden seed from a poorly designed pseudo-random number generator. By taking advantage of known plaintext constraints through gradual candidate testing and branch pruning, the algorithm can reduce a very large search space into a practical recovery process. The experiment successfully recovered both the original seed and plaintext, even still SHA-256 was used to verify its random generation result. Overall, the findings emphasize that simple deterministic generators such as LCG should not be used for real cryptographic protection, but they remain useful for demonstrating backtracking and security weaknesses in an educational setting.

VI. Attachment

[1] Github Repository: https://github.com/Naufal-Pinasthika/Backtracking-LCG

VII. Acknowledgment

The author is deeply thankful to Allah SWT for providing strength, determination, and opportunity to authors in persuading academic journey in Informatics Engineering Institute Teknologi Bandung and Algorithm Strategy paper to completion. The author also expresses deep appreciation to Dr. Ir. Rinaldi Munir, M.T., lecturer on IF2211 Algorithm Strategy course, for his dedicated guidance and support, which has inspired author in many such ways. The author would like to express immense gratitude to the authors family and friends which help authors in their journey.

References

[1] Pasqualini, L., & Parton, M. (2020). Pseudo Random Number Generation: a Reinforcement Learning approach. *Procedia Computer Science*, *170*, 1122–1127. Available: https://arxiv.org/pdf/1912.11531. Accessed Jun. 18, 2026.

[2] Barker, E. (2020). Recommendation for Key Management: Part 1 - General. National Institute of Standards and Technology. Special Publication (NIST SP) 800-57 Pt. 1 Rev. 5. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf. Accessed Jun. 18, 2026.

[3] Barker, E., & Kelsey, J. (2015). Recommendation for Random Number Generation Using Deterministic Random Bit Generators. National Institute of Standards and Technology. Special Publication (NIST SP) 800-90A Rev. 1. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf. Accessed Jun. 18, 2026.

[4] Luo. (2009). Section II: Linear Congruential Generator I. Cornell University Department of Mathematics (Math Explorers Club). Available: https://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html. Accessed Jun. 18, 2026.

[5] National Institute of Standards and Technology. (2015). Secure Hash Standard (SHS). FIPS PUB 180-4. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Accessed Jun. 18, 2026.

[6] Rogaway, P., & Shrimpton, T. (2004). Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. Fast Software Encryption. 3017. 371-388. Available: https://link.springer.com/chapter/10.1007/978-3-540-25937-4_24. Accessed Jun. 18, 2026.

[7] Munir, R. (2026). Algoritma Runut-balik (Backtracking), Bagian 1 dan Bagian 2. Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, STEI ITB. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/15-Algoritma-backtracking-(2026)-Bagian1.pdf, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/26-Program-Dinamis-(2026)-Bagian2.pdf. Accessed Jun. 18, 2026.

[8] Horowitz, E., Sahni, S., & Rajasekaran, S. (2008). Fundamentals of Computer Algorithms, 2nd ed. Universities Press. Available: https://www.universitiespress.com. Accessed Jun. 18, 2026.

[9] Knuth, D. E. (1975). Estimating the Efficiency of Backtrack Programs. Mathematics of Computation. 29. 129. 121–136. Available: https://doi.org/10.1030/S0025-5718-1975-0366113-5. Accessed Jun. 18, 2026.

[10] Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. Cambridge University Press. Available: https://hlevkin.com/hlevkin/60numalgs/C/Press,Teukolsky,%20Vetterling,Flannery-Numerical%20Recipes%20in%20C.pdf. Accessed Jun. 19, 2026.

[11] Boyar, J. (1989). Inferring sequences produced by pseudo-random number generators. Journal of Cryptology. 1. 3. 177–208. Available: https://dl.acm.org/doi/epdf/10.1145/58562.59305. Accessed Jun. 19, 2026.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Anindya Naufal Pinasthika 13524013