Random Number Generators

Colby Endres

January 7, 2026

Random Numbers

Most of cryptography we've seen relies on some notion of "randomness". Examples include:

Random Numbers

Most of cryptography we've seen relies on some notion of "randomness". Examples include:

- Pick a large, *random prime* p (Diffie-Hellman, ElGamal, RSA)

Random Numbers

Most of cryptography we've seen relies on some notion of "randomness". Examples include:

- Pick a large, *random prime* p (Diffie-Hellman, ElGamal, RSA)
- Generate a random *key* (one-time pads)

Random Numbers

Most of cryptography we've seen relies on some notion of "randomness". Examples include:

- Pick a large, *random prime p* (Diffie-Hellman, ElGamal, RSA)
- Generate a random *key* (one-time pads)
- Randomly sample from $(\mathbb{Z}/n\mathbb{Z})^*$ (quadratic nonresidues, Solovay-Strassen, Miller-Rabin)

Random Numbers

Most of cryptography we've seen relies on some notion of "randomness". Examples include:

- Pick a large, *random prime* p (Diffie-Hellman, ElGamal, RSA)
- Generate a random *key* (one-time pads)
- Randomly sample from $(\mathbb{Z}/n\mathbb{Z})^*$ (quadratic nonresidues, Solovay-Strassen, Miller-Rabin)

When implementing these algorithms in practice, how might we generate *quality* random numbers?

Motivation

Is a good source of randomness really that important for cryptography?

Motivation

Is a good source of randomness really that important for cryptography?

- *Yes- a system is only as strong as its weakest link!*

Motivation

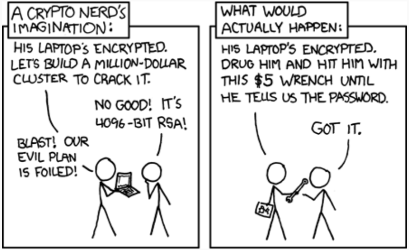
Is a good source of randomness really that important for cryptography?

- *Yes- a system is only as strong as its weakest link!*
- If an adversary could recreate our process for generating random numbers, it stands to reason they can find our private keys

Motivation

Is a good source of randomness really that important for cryptography?

- *Yes- a system is only as strong as its weakest link!*
- If an adversary could recreate our process for generating random numbers, it stands to reason they can find our private keys
- This completely sidesteps the challenge of breaking a cryptographic algorithm!



First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`
  - *"...the pseudo-random generators of this module should not be used for security purposes..."*

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`
  - *"...the pseudo-random generators of this module should not be used for security purposes..."*
- Maybe C++'s `std::random_device()`?

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`
  - *"...the pseudo-random generators of this module should not be used for security purposes..."*
- Maybe C++'s `std::random_device()`?
  - *"...none of these random number engines are cryptographically secure..."*

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`
  - *"...the pseudo-random generators of this module should not be used for security purposes..."*
- Maybe C++'s `std::random_device()`?
  - *"...none of these random number engines are cryptographically secure..."*
- How about Java's `Random` class?

First Steps

Most programming languages provide a random number generator in standard library. Surely we can just use that?

Except...

- Python's `random()`
  - *"...the pseudo-random generators of this module should not be used for security purposes..."*
- Maybe C++'s `std::random_device()`?
  - *"...none of these random number engines are cryptographically secure..."*
- How about Java's `Random` class?
  - *"Instances of java.util.Random are not cryptographically secure..."*

What exactly makes these standard generators inadequate for our purposes?

First Steps

Before discussing this, it's worthwhile to discuss exactly what we want in a random number generator. We will focus on the probability distribution U_N, the uniform distribution with support $\{0, 1, 2, \ldots, N-1\}$.

First Steps

Before discussing this, it's worthwhile to discuss exactly what we want in a random number generator. We will focus on the probability distribution U_N, the uniform distribution with support $\{0, 1, 2, \ldots, N-1\}$. Intuitively, a good random number generator G should obey the following:

First Steps

Before discussing this, it's worthwhile to discuss exactly what we want in a random number generator. We will focus on the probability distribution U_N, the uniform distribution with support $\{0, 1, 2, \ldots, N-1\}$. Intuitively, a good random number generator G should obey the following:

- The distribution of values outputted by G should be "indistinguishable" from U_N
  - Determined by statistical tests (more on this later)

First Steps

Before discussing this, it's worthwhile to discuss exactly what we want in a random number generator. We will focus on the probability distribution U_N, the uniform distribution with support $\{0, 1, 2, \ldots, N-1\}$. Intuitively, a good random number generator G should obey the following:

- The distribution of values outputted by G should be "indistinguishable" from U_N
  - Determined by statistical tests (more on this later)
- If an adversary observes a sequence of values $\{x_i\}$, it should be challenging for them to predict future values *or* recover past values
  - (e.g. 0,1,2,0,1,2,... is uniformly distributed, but easy to predict)

Linear Congruence Generators

One of the simplest methods for generating a random sequence of numbers is the linear congruential generator (LCG).

Linear Congruence Generators

One of the simplest methods for generating a random sequence of numbers is the linear congruential generator (LCG). An LCG is governed by the following recurrence relation:

$$X_k = (a \cdot X_{k-1} + c) \bmod m$$

where the initial value X_0 is referred to as the seed.

Linear Congruence Generators

One of the simplest methods for generating a random sequence of numbers is the linear congruential generator (LCG). An LCG is governed by the following recurrence relation:

$$X_k = (a \cdot X_{k-1} + c) \bmod m$$

where the initial value X_0 is referred to as the seed. For example, if $c = 0$ and $m = 7$, we have the sequences:

$$(5, 3, 6, 5, 3 \dots) \qquad (X_0 = 5, a = 2)$$

$$(5, 1, 3, 2, 6, 4, 5 \dots) \qquad (X_0 = 5, a = 3)$$

Periodicity

These sequences eventually repeat after a set number of iterations.

Periodicity

These sequences eventually repeat after a set number of iterations.

Definition

The period of an LCG is the smallest positive integer n such that $X_{n+\ell} = X_\ell$ for all ℓ.

Periodicity

These sequences eventually repeat after a set number of iterations.

Definition

The period of an LCG is the smallest positive integer n such that $X_{n+\ell} = X_\ell$ for all ℓ.

We would like a period as long as possible. By the Pigeonhole Principle, the period of an LCG is at most m.

Periodicity

These sequences eventually repeat after a set number of iterations.

Definition

The period of an LCG is the smallest positive integer n such that $X_{n+\ell} = X_\ell$ for all ℓ.

We would like a period as long as possible. By the Pigeonhole Principle, the period of an LCG is at most m. As we saw on the previous slide, the value of the period is sensitive to the parameters. We'll attempt to classify the best choices of parameters for maximizing period length.

Classifying Periods

Let L be an LCG. We examine the simplest case first: when $c = 0$.

Classifying Periods

Let L be an LCG. We examine the simplest case first: when $c = 0$.

Theorem

Let a and X_0 be elements of $(\mathbb{Z}/m\mathbb{Z})^*$. Then, the period of L is equal to ord(a).

Classifying Periods

Let L be an LCG. We examine the simplest case first: when $c = 0$.

Theorem

Let a and X_0 be elements of $(\mathbb{Z}/m\mathbb{Z})^*$. Then, the period of L is equal to ord(a).

Proof.

Suppose $X_k \equiv X_0 \bmod m$, where $k > 0$. By unrolling, we see that $X_k \equiv a^k \cdot X_0 \bmod m$. Thus:

Classifying Periods

Let L be an LCG. We examine the simplest case first: when $c = 0$.

Theorem

Let a and X_0 be elements of $(\mathbb{Z}/m\mathbb{Z})^*$. Then, the period of L is equal to ord(a).

Proof.

Suppose $X_k \equiv X_0 \bmod m$, where $k > 0$. By unrolling, we see that $X_k \equiv a^k \cdot X_0 \bmod m$. Thus:

$$a^k X_0 \equiv X_0 \bmod m$$
$$a^k \equiv 1 \bmod m$$

By definition of $(\mathbb{Z}/m\mathbb{Z})^*$, we have that $k = ord(a)$. $\square$

Classifying Periods

Let L be an LCG. We examine the simplest case first: when $c = 0$.

Theorem

Let a and X_0 be elements of $(\mathbb{Z}/m\mathbb{Z})^*$. Then, the period of L is equal to ord(a).

Proof.

Suppose $X_k \equiv X_0 \bmod m$, where $k > 0$. By unrolling, we see that $X_k \equiv a^k \cdot X_0 \bmod m$. Thus:

$$a^k X_0 \equiv X_0 \bmod m$$
$$a^k \equiv 1 \bmod m$$

By definition of $(\mathbb{Z}/m\mathbb{Z})^*$, we have that $k =$ord(a). □

Note that if m is prime, we attain a maximal period of $m - 1$ when taking a to be a generator of $(\mathbb{Z}/m\mathbb{Z})^*$.

Classifying Periods (cont.)

What if $c \neq 0$? To classify such a, we'll use the following lemma:

Classifying Periods (cont.)

What if $c \neq 0$? To classify such a, we'll use the following lemma:

Lemma

For $a \neq 1$, we have $X_k = a^k \cdot X_0 + c \cdot \frac{a^k - 1}{a - 1} \bmod m$

Classifying Periods (cont.)

What if $c \neq 0$? To classify such a, we'll use the following lemma:

Lemma

For $a \neq 1$, we have $X_k = a^k \cdot X_0 + c \cdot \frac{a^k - 1}{a - 1} \bmod m$

Proof.

We argue by induction. The base case of $k = 1$ is clearly true.

Suppose this holds for some n. Then:

Classifying Periods (cont.)

What if $c \neq 0$? To classify such a, we'll use the following lemma:

Lemma

For $a \neq 1$, we have $X_k = a^k \cdot X_0 + c \cdot \frac{a^k - 1}{a - 1} \bmod m$

Proof.

We argue by induction. The base case of $k = 1$ is clearly true. Suppose this holds for some n. Then:

$$\begin{aligned} X_{n+1} &= a \cdot \left(a^n \cdot X_0 + c \cdot \frac{a^n - 1}{a - 1} \right) + c \\ &= a^{n+1} \cdot X_0 + \left(\frac{a - 1}{a - 1} + \frac{a^{n+1} - a}{a - 1} \right) \cdot c \\ &= a^{n+1} \cdot X_0 + \left(\frac{a^{n+1} - 1}{a - 1} \right) \cdot c \end{aligned}$$

□

Classifying Periods (cont.)

This allows us to prove the following when $a \neq 1$:

Classifying Periods (cont.)

This allows us to prove the following when $a \neq 1$:

Theorem

Let m be prime and $X_0 \neq -\frac{c}{a-1}$, L has period ord(a) for any value c.

Classifying Periods (cont.)

This allows us to prove the following when $a \neq 1$:

Theorem

Let m be prime and $X_0 \neq -\frac{c}{a-1}$, L has period ord(a) for any value c.

Proof.

Suppose $X_k = X_0$. Using the previous lemma:

$$a^k \cdot X_0 + c \cdot \frac{a^k - 1}{a - 1} \equiv X_0 \bmod m$$
$$(a^k - 1)\left(X_0 + \frac{c}{a-1}\right) \equiv 0 \bmod m$$

Classifying Periods (cont.)

This allows us to prove the following when $a \neq 1$:

Theorem

Let m be prime and $X_0 \neq -\frac{c}{a-1}$, L has period ord(a) for any value c.

Proof.

Suppose $X_k = X_0$. Using the previous lemma:

$$a^k \cdot X_0 + c \cdot \frac{a^k - 1}{a - 1} \equiv X_0 \bmod m$$

$$(a^k - 1)\left(X_0 + \frac{c}{a-1}\right) \equiv 0 \bmod m$$

Since $(\mathbb{Z}/m\mathbb{Z})^*$ is an integral domain, we must have $a^k \equiv 1 \bmod m$, giving us a period of $\mathrm{ord}(a)$. □

LCG in Practice

Determining the period length of L for arbitrary choices of a, c, m and X_0 is more challenging. However, we have the following result from Hull and Dobell:

LCG in Practice

Determining the period length of L for arbitrary choices of a, c, m and X_0 is more challenging. However, we have the following result from Hull and Dobell:

Theorem

An LCG has full period m if:

- *$gcd(m, c)$=1*
- *$a - 1$ is divisible by all prime factors of m*
- *If $m \equiv 0 \bmod 4$, then $a \equiv 1 \bmod 4$.*

LCG in Practice

Determining the period length of L for arbitrary choices of a, c, m and X_0 is more challenging. However, we have the following result from Hull and Dobell:

Theorem

An LCG has full period m if:

- *gcd(m, c)=1*
- *$a - 1$ is divisible by all prime factors of m*
- *If $m \equiv 0 \bmod 4$, then $a \equiv 1 \bmod 4$.*

If we follow these criteria, we can create a sequence with a period as long as we like. The question is: how secure is this?

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Idea

Denote $d_k = x_{k+1} - x_k$.

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Idea

Denote $d_k = x_{k+1} - x_k$. A simple induction shows that $d_{k+j} = a^j \cdot d_k$.

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Idea

Denote $d_k = x_{k+1} - x_k$. A simple induction shows that $d_{k+j} = a^j \cdot d_k$. It follows that $d_{k+1}^2 - d_k \cdot d_{k+2} \equiv 0 \bmod m$. Compute this for the $n-3$ eligible candidates and call this set D.

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Idea

Denote $d_k = x_{k+1} - x_k$. A simple induction shows that $d_{k+j} = a^j \cdot d_k$. It follows that $d_{k+1}^2 - d_k \cdot d_{k+2} \equiv 0 \bmod m$. Compute this for the $n-3$ eligible candidates and call this set D. For an arbitrary pair $(u, v) \in D$, we have $\gcd(u, v) = m$ with probability $\frac{6}{\pi^2}$. Thus:

Cracking the LCG

Unfortunately, an adversary can *completely determine the state of an LCG*, solely by observing consecutive outputs. We begin by discussing how to recover the unknown modulus m by observing n consecutive outputs.

Idea

Denote $d_k = x_{k+1} - x_k$. A simple induction shows that $d_{k+j} = a^j \cdot d_k$. It follows that $d_{k+1}^2 - d_k \cdot d_{k+2} \equiv 0 \bmod m$. Compute this for the $n-3$ eligible candidates and call this set D. For an arbitrary pair $(u, v) \in D$, we have $\gcd(u, v) = m$ with probability $\frac{6}{\pi^2}$. Thus:

$$\mathbb{P}(\text{fail to find } m) \leq \prod_{(u,v) \in D} \mathbb{P}(\gcd(u, v) \neq m) \leq \left(1 - \frac{6}{\pi^2}\right)^{\binom{n-3}{2}}$$

which vanishes as $n \to \infty$

Cracking the LCG (cont.)

With the modulus m found, determining the rest of the state is trivial.

Cracking the LCG (cont.)

With the modulus m found, determining the rest of the state is trivial. Assuming our opponent observes X_{i-1}, X_i, X_{i+1}, they merely need to solve the following 2×2 system in $\mathbb{Z}/m\mathbb{Z}$:

$$X_{i+1} \equiv aX_i + c \bmod m$$
$$X_i \equiv aX_{i-1} + c \bmod m$$

Cracking the LCG (cont.)

With the modulus m found, determining the rest of the state is trivial. Assuming our opponent observes X_{i-1}, X_i, X_{i+1}, they merely need to solve the following 2×2 system in $\mathbb{Z}/m\mathbb{Z}$:

$$\begin{aligned} X_{i+1} &\equiv aX_i + c \bmod m \\ X_i &\equiv aX_{i-1} + c \bmod m \end{aligned}$$

Our system is now *completely compromised*, as an attacker can both recover previous values and predict future ones!

Case Study: randq1

- Many programming languages use some form of LCG for generating random integers
- A popular choice is `randq1`, with the following parameters:
  - $m = 2^{32}, a = 1664525, c = 1013904223$
- Running the previous algorithm has a near perfect chance at recovering m, regardless of starting seed

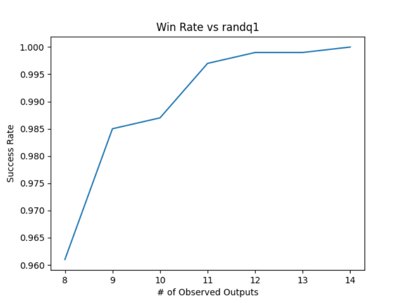


Figure 1: Average Win rate against `randq1`

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values.

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values. What if we cooked up a RNG with a more complicated state?

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values. What if we cooked up a RNG with a more complicated state?

- Would require more bookkeeping, but should be harder to crack

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values. What if we cooked up a RNG with a more complicated state?

- Would require more bookkeeping, but should be harder to crack
- We'll discuss the Mersenne Twister, developed in 1997 as an improvement over the standard LCG

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values. What if we cooked up a RNG with a more complicated state?

- Would require more bookkeeping, but should be harder to crack
- We'll discuss the Mersenne Twister, developed in 1997 as an improvement over the standard LCG
- It's currently the de facto RNG for many programming languages

Towards Better RNGs

The Achilles' heel of the LCG is its small state. To predict the next value, an enemy needs only to determine four values. What if we cooked up a RNG with a more complicated state?

- Would require more bookkeeping, but should be harder to crack
- We'll discuss the Mersenne Twister, developed in 1997 as an improvement over the standard LCG
- It's currently the de facto RNG for many programming languages

As the name suggests, the twister relies on Mersenne primes (i.e. primes of the form $2^n - 1$. The most common variant is MT19937, which generates a random unsigned 32 bit integer

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

1. Initialization
    - MT19937 keeps a 624-element state vector

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

1. Initialization
   - MT19937 keeps a 624-element state vector
   - The remaining 623 elements are generated by the modified LCG:

$$x_i = (f \cdot (x_{i-1} \oplus (x_{i-1} \gg 30)) + i) \bmod 2^{32}$$

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

1. Initialization
   - MT19937 keeps a 624-element state vector
   - The remaining 623 elements are generated by the modified LCG:

$$x_i = (f \cdot (x_{i-1} \oplus (x_{i-1} \gg 30)) + i) \bmod 2^{32}$$

2. Tempering
   - Before being provided to the user, a state element is tempered to guarantee some nice statistical properties

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

1. Initialization
   - MT19937 keeps a 624-element state vector
   - The remaining 623 elements are generated by the modified LCG:

$$x_i = (f \cdot (x_{i-1} \oplus (x_{i-1} \gg 30)) + i) \bmod 2^{32}$$

2. Tempering
   - Before being provided to the user, a state element is tempered to guarantee some nice statistical properties
   - Perform bitwise operations with “magic numbers”

MT19937, At a Glance

The Mersenne Twister consists of three stages. Like earlier, we start with some seed x_0.

1. Initialization
    - MT19937 keeps a 624-element state vector
    - The remaining 623 elements are generated by the modified LCG:

$$x_i = (f \cdot (x_{i-1} \oplus (x_{i-1} \gg 30)) + i) \bmod 2^{32}$$

2. Tempering
    - Before being provided to the user, a state element is tempered to guarantee some nice statistical properties
    - Perform bitwise operations with “magic numbers”
    - These constants are experimentally chosen to pass statistical randomness tests

MT19937, At a Glance (cont.)

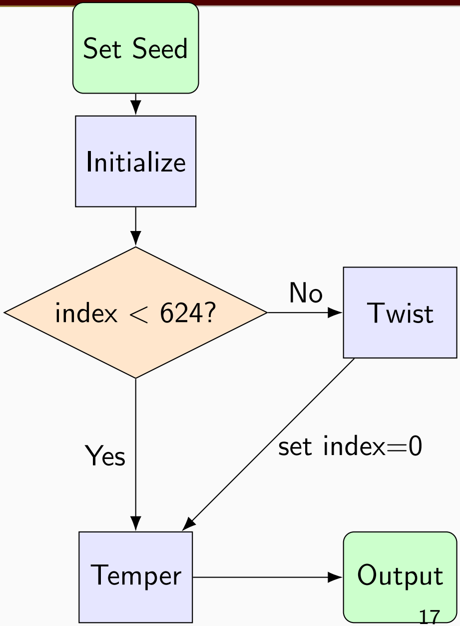
3. Twisting

MT19937, At a Glance (cont.)

3. Twisting
   - Once all 624 elements have been used, we must *twist* the state vector to get new numbers

MT19937, At a Glance (cont.)

3. Twisting
   - Once all 624 elements have been used, we must *twist* the state vector to get new numbers
   - Essentially amounts to a matrix multiplication in $\mathbb{F}_2$, along with more bit operations



One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator
  - Bit operations are incredibly fast for computers

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator
  - Bit operations are incredibly fast for computers
- Equidistribution in higher dimensions

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator
  - Bit operations are incredibly fast for computers
- Equidistribution in higher dimensions
  - All 2^{32} bit combinations are equally likely to appear in a given period

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator
  - Bit operations are incredibly fast for computers
- Equidistribution in higher dimensions
  - All 2^{32} bit combinations are equally likely to appear in a given period
  - LCG's have notoriously bad performance in this (more on this later)

One Step Forward...

So what do we get out of the construction?

- Enormous period ($2^{19937} - 1$)
- Relatively efficient generator
  - Bit operations are incredibly fast for computers
- Equidistribution in higher dimensions
  - All 2^{32} bit combinations are equally likely to appear in a given period
  - LCG's have notoriously bad performance in this (more on this later)
- Passes a large amount of statistical randomness tests

Two Steps Back

MT19937 looks like a great RNG, but will it work for cryptography?

Two Steps Back

MT19937 looks like a great RNG, but will it work for cryptography? The answer, unfortunately, is no. Why?

- Suppose an adversary observes 624 random elements and knows that MT19937 is being employed

Two Steps Back

MT19937 looks like a great RNG, but will it work for cryptography? The answer, unfortunately, is no. Why?

- Suppose an adversary observes 624 random elements and knows that MT19937 is being employed
- The temper and twist steps can be phrased as linear transformations in $\mathbb{F}_2^{32}$

Two Steps Back

MT19937 looks like a great RNG, but will it work for cryptography? The answer, unfortunately, is no. Why?

- Suppose an adversary observes 624 random elements and knows that MT19937 is being employed
- The temper and twist steps can be phrased as linear transformations in $\mathbb{F}_2^{32}$
  - *These matrices are invertible!*

Two Steps Back

MT19937 looks like a great RNG, but will it work for cryptography? The answer, unfortunately, is no. Why?

- Suppose an adversary observes 624 random elements and knows that MT19937 is being employed
- The temper and twist steps can be phrased as linear transformations in $\mathbb{F}_2^{32}$
  - *These matrices are invertible!*
- If our opponent calculates the inverse matrices associated with the transformations (which are the same for all instances of MT19937), they can recover the state vector

As with our LCG, we lack both forwards and backwards security, a massive problem for security

A Step Away

At their core, both the LCG and Mersenne Twister suffer from the same problem: determinism

A Step Away

At their core, both the LCG and Mersenne Twister suffer from the same problem: determinism

- Assuming the seed doesn't change, the generator will output the exact same sequence every time

A Step Away

At their core, both the LCG and Mersenne Twister suffer from the same problem: determinism

- Assuming the seed doesn't change, the generator will output the exact same sequence every time
  - Good for reproducibility and debugging, bad for cryptography
- These generators are referred to as pseudorandom, due to their determinism

A Step Away

At their core, both the LCG and Mersenne Twister suffer from the same problem: determinism

- Assuming the seed doesn't change, the generator will output the exact same sequence every time
  - Good for reproducibility and debugging, bad for cryptography
- These generators are referred to as pseudorandom, due to their determinism
  - *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" - von Neumann*

A Step Away

At their core, both the LCG and Mersenne Twister suffer from the same problem: determinism

- Assuming the seed doesn't change, the generator will output the exact same sequence every time
  - Good for reproducibility and debugging, bad for cryptography
- These generators are referred to as pseudorandom, due to their determinism
  - *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" - von Neumann*

If pure mathematical techniques can't cut it, where might we get a source of "true randomness"?

Physical Randomness

The answer lies, not in mathematics, but in the physical world. Many processes exhibit random (or at the very least hard to predict) behavior. Examples include:

Physical Randomness

The answer lies, not in mathematics, but in the physical world. Many processes exhibit random (or at the very least hard to predict) behavior. Examples include:

- Physical environment

Physical Randomness

The answer lies, not in mathematics, but in the physical world. Many processes exhibit random (or at the very least hard to predict) behavior. Examples include:

- Physical environment
    - Atmospheric noise, nuclear decay, various quantum phenomena

Physical Randomness

The answer lies, not in mathematics, but in the physical world. Many processes exhibit random (or at the very least hard to predict) behavior. Examples include:

- Physical environment
  - Atmospheric noise, nuclear decay, various quantum phenomena
- Electronic hardware
  - Clock jitter, OS interrupts, disk reads

Physical Randomness

The answer lies, not in mathematics, but in the physical world. Many processes exhibit random (or at the very least hard to predict) behavior. Examples include:

- Physical environment
  - Atmospheric noise, nuclear decay, various quantum phenomena
- Electronic hardware
  - Clock jitter, OS interrupts, disk reads
- You
  - Mouse clicks, keyboard timing, cursor velocity

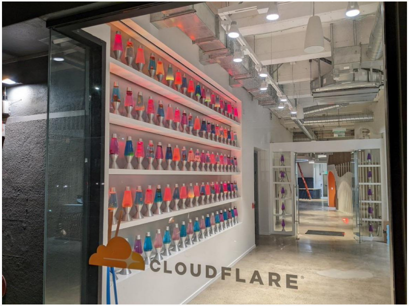


Figure 2: Cloudflare's "lamp wall" used in generating SSL keys

TRNGs

Generators that rely on these phenomena are referred to as true random number generators (TRNGS)

- All TRNGs rely on a quality source of entropy
- The key is that the underlying phenomenon should be impossible to predict, even if your adversary knows what the source is

We'll discuss how your computer (probably) generates random numbers for cryptographic applications.

Case Study: `/dev/random`

If you're on a Unix-based system, your operating system generates true random numbers via the device file `/dev/random`. In a nutshell,

1. Entropy Pooling

Case Study: `/dev/random`

If you're on a Unix-based system, your operating system generates true random numbers via the device file `/dev/random`. In a nutshell,

1. Entropy Pooling
   - Kernel queries various sources of randomness and stores the resulting bits in an input buffer
   - Each event is given an estimate for how many bits of entropy are produced
2. Pool Mixing

Case Study: `/dev/random`

If you're on a Unix-based system, your operating system generates true random numbers via the device file `/dev/random`. In a nutshell,

1. Entropy Pooling
    - Kernel queries various sources of randomness and stores the resulting bits in an input buffer
    - Each event is given an estimate for how many bits of entropy are produced
2. Pool Mixing
    - These bits are then mixed into the pool using some cryptographic primitive
        - Older versions of Linux use the SHA-1 hash of the bitstring
        - Modern kernels have migrated to the ChaCha20 stream cipher
3. Number Generation

Case Study: `/dev/random`

If you're on a Unix-based system, your operating system generates true random numbers via the device file `/dev/random`. In a nutshell,

1. Entropy Pooling
    - Kernel queries various sources of randomness and stores the resulting bits in an input buffer
    - Each event is given an estimate for how many bits of entropy are produced
2. Pool Mixing
    - These bits are then mixed into the pool using some cryptographic primitive
        - Older versions of Linux use the SHA-1 hash of the bitstring
        - Modern kernels have migrated to the ChaCha20 stream cipher
3. Number Generation
    - Initialize a good PRNG with a portion of the entropy pool
    - Return the number(s) and decrease the amount of entropy

Statistical Testing

We've talked at length about the security properties of each RNG. Recall that we also want some notion of evenness, since this is supposed to represent the uniform distribution. A common technique is the χ^2 goodness-of-fit test.

Statistical Testing

We've talked at length about the security properties of each RNG. Recall that we also want some notion of evenness, since this is supposed to represent the uniform distribution. A common technique is the χ^2 goodness-of-fit test.

- Generate 10^6 numbers in $\{0, 1, \ldots, 2^{32} - 1\}$

Statistical Testing

We've talked at length about the security properties of each RNG. Recall that we also want some notion of evenness, since this is supposed to represent the uniform distribution. A common technique is the χ^2 goodness-of-fit test.

- Generate 10^6 numbers in $\{0, 1, \ldots, 2^{32} - 1\}$
- Assume that G gives a uniform distribution as null hypothesis

Statistical Testing

We've talked at length about the security properties of each RNG. Recall that we also want some notion of evenness, since this is supposed to represent the uniform distribution. A common technique is the χ^2 goodness-of-fit test.

- Generate 10^6 numbers in $\{0, 1, \ldots, 2^{32} - 1\}$
- Assume that G gives a uniform distribution as null hypothesis
- Experimental evidence fails to reject H_0 at standard $p < 0.05$
  - But very sensitive to starting seed

Generator	χ^2	p
randq	237.3	0.7791
MT19937	245.2	0.6584
urandom	245.5	0.6530

Table 1: χ^2 test with $\nu = 255$ degrees of freedom

The NIST Standard

Of course, there are more tests than just goodness-of-fit. NIST has a battery of statistical tests that are used in evaluating RNGs. Some notes about our three candidates:

- LCG (185/187)
  - Abject failure in the spectral test
  - Marsaglia showed that outputs of an LCG do not uniformly populate the unit cube, but instead lie on hyperplanes
- Both MT19937 and `/dev/random` pass all statistical tests
  - `/dev/random` has slightly more uniform p values

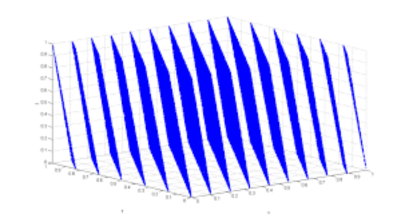


Figure 3: Spectral test on IBM's RANDU