

# Cracking Chaos

Making, Using, and Breaking PRNGs

By 1nfocalypse

# What is “Random”: Some Definitions

Randomness is a bit hard to rigorously define, especially in the context of deterministic machines. But, we do our best.

- True Randomness: Entropy
- The Random Oracle Model
- Statistical Randomness

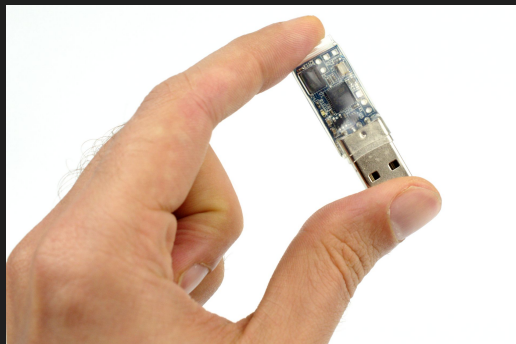


"The Hangover"

# True Randomness

True randomness is an essential but expensive commodity on computers.

- System Entropy
- HRNGs and dedicated tooling
- Why do we care? Statistics and hardness assertions.



# Insecure PRNGs

Sometimes, we only care about the speed and statistical quality of outputs.

In such cases, the promise of “n-bit” security usually doesn’t quite hold up.

- CVE 2008-0166, 7.5: Debian OpenSSL - Weak entropy
- CVE 2015-5267, 7.5: Moodle - MT19937 Password Recovery

## CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Weakness ID: 338  
Vulnerability Mapping: **ALLOWED**  
Abstraction: Base

View customized information:

Conceptual

Operational

Mapping  
Friendly

Complete

Custom

### Description

The product uses a Pseudo-Random Number Generator (PRNG) in a security context, but the PRNG's algorithm is not cryptographically strong.

### Extended Description

When a non-cryptographic PRNG is used in a cryptographic context, it can expose the cryptography to certain types of attacks.

Often a pseudo-random number generator (PRNG) is not designed for cryptography. Sometimes a mediocre source of randomness is sufficient or preferable for algorithms that use random numbers. Weak generators generally take less processing power and/or do not use the precious, finite, entropy sources on a system. While such PRNGs might have very useful features, these same features could be used to break the cryptography.

# Linear Congruential Generators

An example of an early, insecure PRNG is the Linear Congruential Generator (LCG). Extremely simple and fairly fast, these generators were prolific in early computing, and can most readily be found in legacy or embedded systems. The maximal period is typically considered to be  $m$ .

$$X_{n+1} = (a \cdot x_n + c) \bmod m$$

1.  $c=0, m \in \mathbb{P}$  (Lehmer Form,  $m-1$  period, slow)
2.  $c=0, m \in \{2^n \mid n \in \mathbb{N}\}$  (Non-maximal period, fast)
3.  $c \neq 0, m \in \{2^n \mid n \in \mathbb{N}\}$  (Maximal period and fast)

# Linear Congruential Generators

An example of an early, insecure PRNG is the Linear Congruential Generator (LCG). Extremely simple and fairly fast, these generators were prolific in early computing, and can most readily be found in legacy or embedded systems. The maximal period is typically considered to be  $m$ .

$$X_{n+1} = (a \cdot x_n + c) \bmod m$$

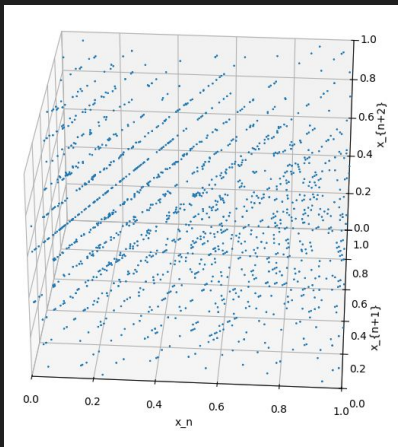
1.  $c=0, m \in \mathbb{P}$  (Lehmer Form,  $m-1$  period, slow)
2.  $c=0, m \in \{2^n \mid n \in \mathbb{N}\}$  (Non-maximal period, fast)
3.  $c \neq 0, m \in \{2^n \mid n \in \mathbb{N}\}$  (Maximal period and fast)



# LCGs: Flaws and Detection

LCGs are, unfortunately, not statistically sufficient for use.

- Detectable by Spectral Test Failure
- Marsaglia's Theorem



## LCGs: Inversion

By the same reasons they are statistically insufficient, they can be broken.

We will walk through a general LCG break, then demonstrate a computational break of *Numerical Recipes'* **ranqd1** procedure.



## LCGs: Inversion

Given several outputs of a LCG,  $O_n$ , form pseudo-Hankel matrices of the form:

$O_n$	$O_{n+1}$	1
$O_{n+1}$	$O_{n+2}$	1
$O_{n+2}$	$O_{n+3}$	1

$O_{n+1}$	$O_{n+2}$	1
$O_{n+2}$	$O_{n+3}$	1
$O_{n+3}$	$O_{n+4}$	1

Take the determinants  $D_1$ ,  $D_2$ , and find  $\text{GCD}(D_1, D_2)$ .

This gives a multiple of the modulus, which can typically be inferred directly by factorization or can be directly enumerated by repeating this operation with more outputs.

## LCGs: Inversion

After enumerating our modulus  $M$ , we next seek the multiplicand  $a$ . In the case that  $\text{GCD}(o_2 - o_1, m) = 1$ ,  
 $a = (o_3 - o_2)(o_2 - o_1)^{-1} \bmod m$ .

Otherwise, we must solve a linear Diophantine equation:

$$ax - ny = b$$

We will pursue the case of  $\text{GCD}(o_2 - o_1, m) = 1$ . Once  $a, m$  have been enumerated, finding  $c$  is as simple as solving:

$$o_2 = a \cdot o_1 + c \bmod m$$

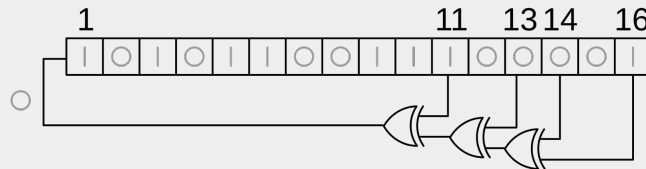
# LCGs: ranqd1 Break Demo and Visualization of Marsaglia's Theorem

# Linear Feedback Shift Registers

Another classic, the LFSR, is a linear generator that makes use of bitwise arithmetic for fast pseudo-random bits. They are often found in embedded environments.

Forms:

- Fibonacci
- Galois
- XOR-Shift

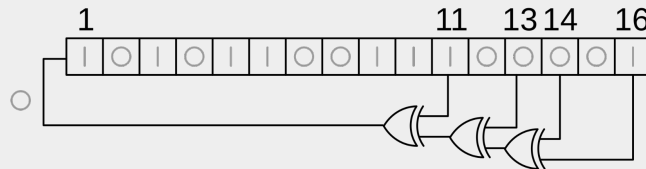


# Linear Feedback Shift Registers

Another classic, the LFSR, is a linear generator that makes use of bitwise arithmetic for fast pseudo-random bits. They are often found in embedded environments.

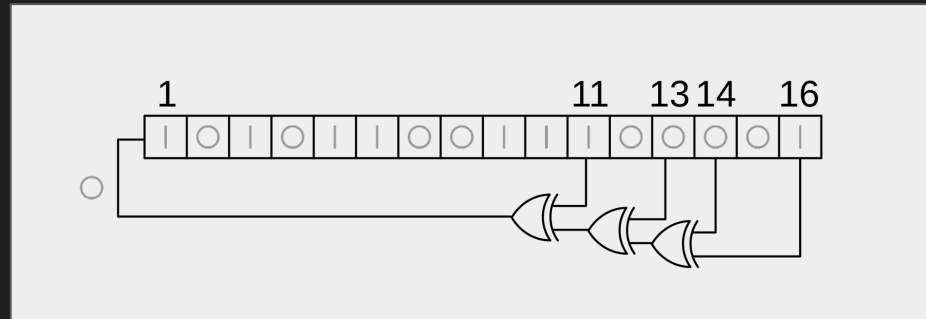
Forms:

- Fibonacci ←
- Galois
- XOR-Shift



## LFSRs: Fibonacci

Fibonacci Generators work by selecting inputs, or “taps”, corresponding to a binary polynomial. If said polynomial is primitive in the given field, the period will be maximal, equal to  $2^n - 1$ , where  $n$  is the degree of the feedback polynomial.



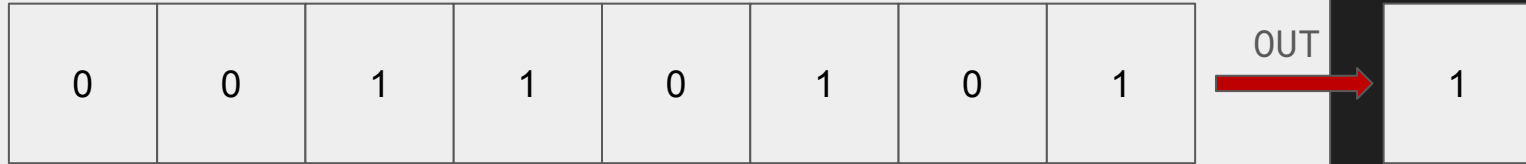
# LFSRs: Fibonacci

$$x^7+x^6+1$$

0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

# LFSRs: Fibonacci

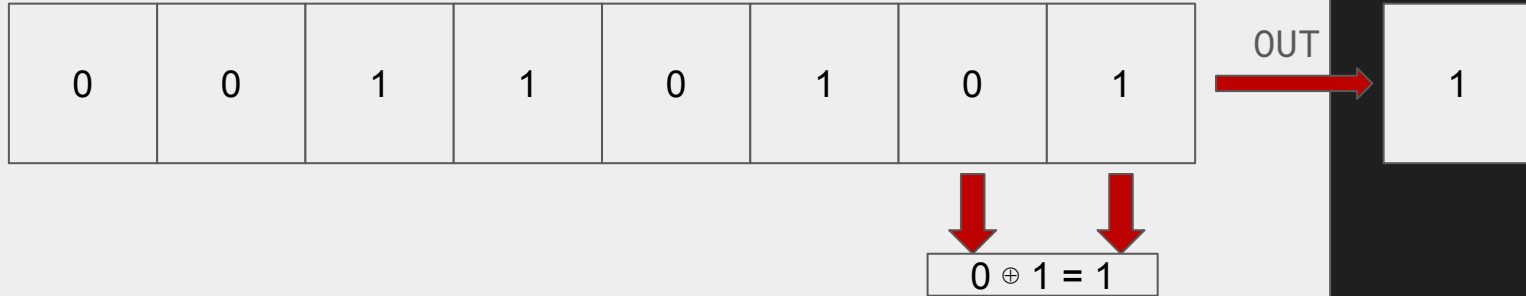
$$x^7 + x^6 + 1$$





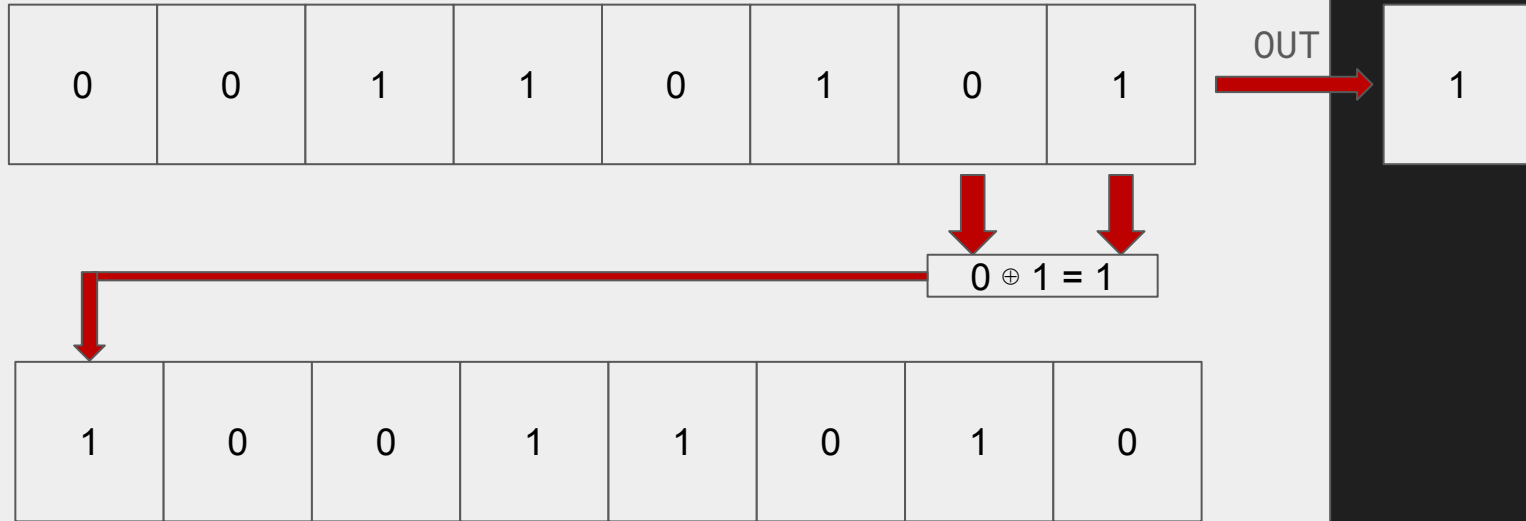
# LFSRs: Fibonacci

$$x^7 + x^6 + 1$$



# LFSRs: Fibonacci

$$x^7 + x^6 + 1$$



# LFSRs: Flaws and Detection

LFSRs notably fail two tests:

- Binary Matrix Rank Test
- Linearity Test
- Former is more distinguishing

Inversion:

- Berlekamp-Massey

It's often more prudent to just try BM.

# Mersenne Twisters

Mersenne Twisters, particularly the MT19937 variant, are one of the most common PRNGs in use.

- Built off of Generalized Feedback Shift Registers (GFSRs)
- Specifically, Twisted Generalized Feedback Shift Registers (Rationalized), or TGFSR(R)s.
- State is generated by the below relation, which is then tempered and output.

$$x_{k+n} = x_{k+m} \oplus (x_k^u / x_{k+1}^l) A$$

# MT19937: A Popular Variant

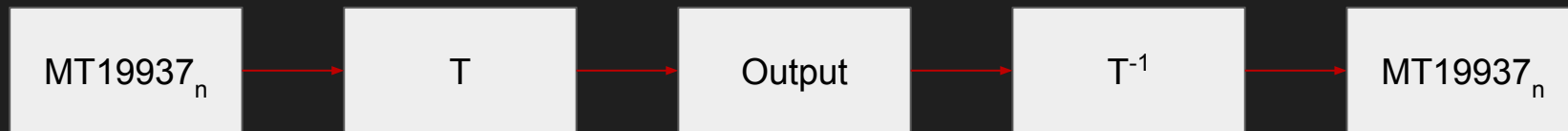
- High quality PRNG, albeit somewhat slow
  - Period:  $2^{19937}-1$
  - State Width: 624 32 bit words
  - Equidistributed up to 624 dimensions after tempering, deteriorates afterwards.

Table II. Parameters and  $k$  Distribution of Mersenne Twister

Generator		The order of equidistribution					
MT19937    (135)	$(w, n, m, r) = (32, 624, 397, 31)$	19937	9968	6240	4984	3738	3115
	$a = 9908B0DF$	2493	2492	1869	1869	1248	1246
	$u = 11$	1246	1246	1246	1246	623	623
	$s = 7, b = 9D2C5680$	623	623	623	623	623	623
	$t = 15, c = EFC60000$	623	623	623	623	623	623
	$l = 18$	623	623				

# MT19937: Flaws and Detection

- Detection is not easy, and is parameter-dependent
- MT19937 fails equidistribution past dim 623
- “Warm up” period required with poor seed choice in original variant.
- Inversion:  
The temper operation is equivalent to matrix multiplication by a non-singular matrix  $T$ , thus meaning it is invertible. Thus, with 624 outputs, the state of MT19937 can be reconstructed by applying the inverse tempering matrix to the outputs, enabling state reconstruction. With some seeding operations, less outputs are required.



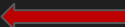
# Counter-Based PRNGs

A relatively new class of PRNG, counter-based PRNGs take inspiration from block ciphers in counter mode.

- Use weakened cryptographic primitives
- Extremely fast with massive parallelization opportunities
- Most common reference is the Random123 library by D.E. Shaw
  - ARS
  - Threefry
  - Philox
- Used in cuRAND, C++26, NumPy, Intel OneAPI, and more
- Random123 generators all pass TestU01's BigCrush
- Extremely long periods (at least  $2^{128}$  per D.E. Shaw)
- No claims of cryptographic security

# Counter-Based PRNGs

A relatively new class of PRNG, counter-based PRNGs take inspiration from block ciphers in counter mode.

- Use weakened cryptographic primitives
- Extremely fast with massive parallelization opportunities
- Most common reference is the Random123 library by D.E. Shaw
  - ARS
  - Threefry
  - Philox 
- Used in cuRAND, C++26, NumPy, Intel OneAPI, and more
- Random123 generators all pass TestU01's BigCrush
- Extremely long periods (at least  $2^{128}$  per D.E. Shaw)
- No claims of cryptographic security

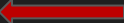


# CBPRNGs: Philox

- Completely original product of Random123
- Hybrid Feistel/Substitution structure
- Multiple forms:
  - Philox2x64-10/6
  - Philox4x32-10/7
  - Philox4x64-10/7
- All official variants pass BigCrush
- Most common generator from Random123

```
+ template<class _UIntType,  
+         size_t __w, size_t __n,  
+         size_t __r, _UIntType... __consts>  
+ void  
+ std::philox_engine<_UIntType, __w, __n, __r, __consts...>::_philox()  
+ {  
+     std::array<_UIntType, __n> __outputSeq{};  
+     for (size_t __j = 0; __j < __n; ++__j)  
+     {  
+         __outputSeq[__j] = __x[__j];  
+     }  
+     for (unsigned long __i = 0; __i < __r; ++__i)  
+     {  
+         std::array<_UIntType, __n> __intermedSeq{};  
+         if (__n == 4)  
+         {  
+             __intermedSeq[0] = __outputSeq[2];  
+             __intermedSeq[1] = __outputSeq[1];  
+             __intermedSeq[2] = __outputSeq[0];  
+             __intermedSeq[3] = __outputSeq[3];  
+         } else  
+         {  
+             __intermedSeq[0] = __outputSeq[0];  
+             __intermedSeq[1] = __outputSeq[1];  
+         }  
+         for (unsigned long __k = 0; __k < (__n/2); ++__k)  
+         {  
+             __outputSeq[2*__k] = __mulhi(__intermedSeq[2*__k], multipliers[__k])  
+             ^ (((this->__k[__k] + (__j * round_consts[__k])) & this->max()))  
+             ^ __intermedSeq[2*__k+1];  
+             __outputSeq[(2*__k)+1] = __mullo(__intermedSeq[2*__k],  
+             multipliers[__k]);  
+         }  
+     }  
+     for (unsigned long __j = 0; __j < __n; ++__j)  
+     {  
+         __y[__j] = __outputSeq[__j];  
+     }  
+ }
```

# CBPRNGs: Philox

- Completely original product of Random123
- Hybrid Feistel/Substitution structure
- Multiple forms:
  - Philox2x64-10/6
  - Philox4x32-10/7 
  - Philox4x64-10/7
- All official variants pass BigCrush
- Most common generator from Random123

```
+ template<class _UIntType,  
+   size_t __w, size_t __n,  
+   size_t __r, _UIntType... __consts>  
+ void  
+ std::philox_engine<_UIntType, __w, __n, __r, __consts...>::_philox()  
+ {  
+   std::array<_UIntType, __n> __outputSeq{};  
+   for (size_t __j = 0; __j < __n; ++__j)  
+   {  
+     __outputSeq[__j] = __x[__j];  
+   }  
+   for (unsigned long __j = 0; __j < __r; ++__j)  
+   {  
+     std::array<_UIntType, __n> __intermedSeq{};  
+     if (__n == 4)  
+     {  
+       __intermedSeq[0] = __outputSeq[2];  
+       __intermedSeq[1] = __outputSeq[1];  
+       __intermedSeq[2] = __outputSeq[0];  
+       __intermedSeq[3] = __outputSeq[3];  
+     } else  
+     {  
+       __intermedSeq[0] = __outputSeq[0];  
+       __intermedSeq[1] = __outputSeq[1];  
+     }  
+     for (unsigned long __k = 0; __k < (__n/2); ++__k)  
+     {  
+       __outputSeq[2*__k] = __mulhi(__intermedSeq[2*__k], multipliers[__k])  
+         ^ (((this->__k[__k] + (__j * round_consts[__k])) & this->max()))  
+         ^ __intermedSeq[2*__k+1];  
+       __outputSeq[(2*__k)+1] = __mullo(__intermedSeq[2*__k],  
+         multipliers[__k]);  
+     }  
+   }  
+   for (unsigned long __j = 0; __j < __n; ++__j)  
+   {  
+     __y[__j] = __outputSeq[__j];  
+   }  
+ }
```

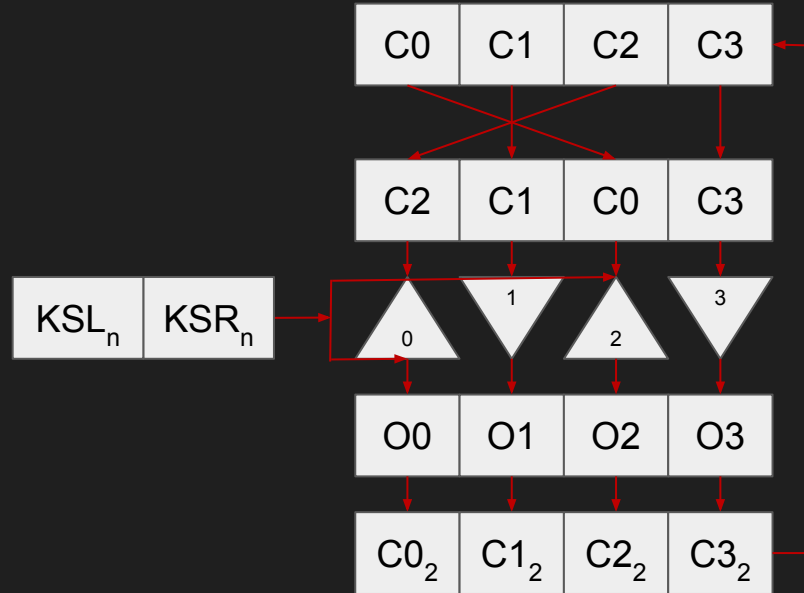
# CBPRNGs: Philox4x32-10

$\text{mlo}(a,b) \rightarrow ab \bmod 2^{32}$

$\text{mhi}(a,b) \rightarrow ab \gg 32$

$\text{KSL}_n \rightarrow \text{LK} + (n \cdot R[0]) \bmod 2^{32}$

$\text{KSR}_n \rightarrow \text{RK} + (n \cdot R[1]) \bmod 2^{32}$



# Philox4x32-10: Flaws and Detection

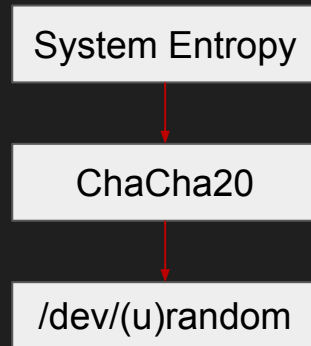
- No statistical differentiator is known at the time of writing.
- No break is known at the time of writing.
- This does not mean Philox (or any CBPRNG) is secure.
- There is no published cryptanalysis of these generators.



# Secure PRNGs

For cryptographic use, we have CSPRNGs (or in NIST verbiage, DRBGs).

These give us high quality pseudo-random numbers with hardness assertions - they are explicitly designed to be hard to invert or otherwise predict to keep n-bit security.



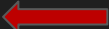
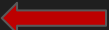
# Stream Ciphers

Probably the most recognizable CSPRNGs are stream ciphers

- Keystreams are instead used as pseudorandom numbers
- If the keystream is predictable, the data isn't secure
- Typically very fast
- Examples:
  - ChaCha20
  - RC4
  - Rabbit

# Stream Ciphers

Probably the most recognizable CSPRNGs are stream ciphers

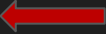
- Keystreams are instead used as pseudorandom numbers
- If the keystream is predictable, the data isn't secure
- Typically very fast
- Examples:
  - ChaCha20 
  - RC4 
  - Rabbit

## RC4: What Not To Do

- RSA Security's "Rivest Cipher 4" (AKA Ron's Code 4)
  - Source never formally released, but leaked
  - Multiple issues found
    - Second byte bias
    - Key schedule weaknesses
  - Use is disallowed in TLS because of flaws (RFC 7465)
  - Was historically also used as a PRNG



## RC4: What Not To Do

- RSA Security's "Rivest Cipher 4" (AKA Ron's Code 4)
  - Source never formally released, but leaked
  - Multiple issues found
    - Second byte bias 
    - Key schedule weaknesses
  - Use is disallowed in TLS because of flaws (RFC 7465)
  - Was historically also used as a PRNG

# RC4: A Brief Description

Initialization:

$i = 0$

$j = 0$

Generation loop:

$i = i + 1$

$j = j + S[i]$

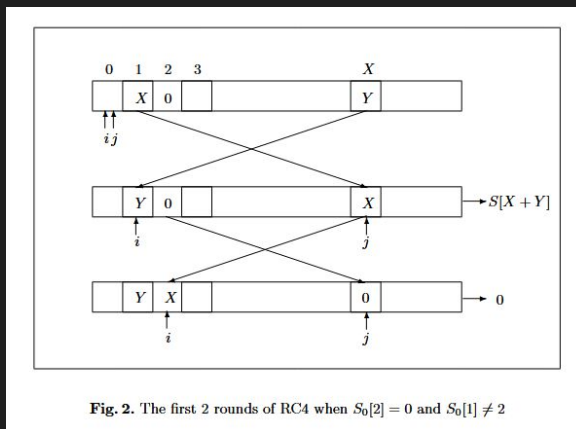
Swap( $S[i], S[j]$ )

Output  $z = S[S[i] + S[j]]$

# RC4: Second Byte Bias

RC4 can be distinguished with good probability very quickly.

- Polynomial sampling
- 60+% correct distinguisher
- Can this be mitigated? Should we use a mitigated version?



## RC4: Second Byte Bias Demo

# ChaCha20

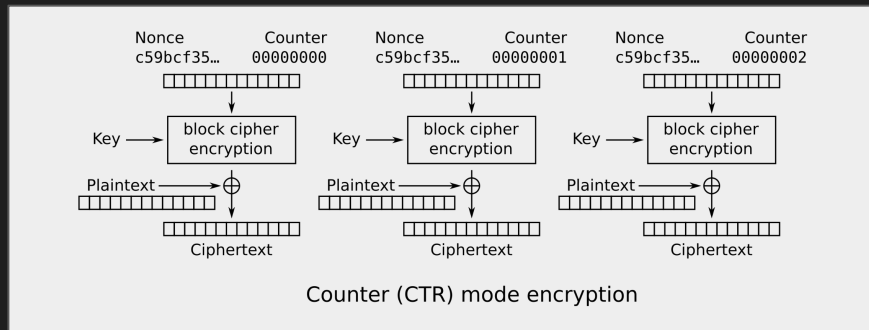
ChaCha20 is a modern, ARX-based stream cipher.

- Developed by Daniel J. Bernstein to succeed Salsa20
- Emphasis on side-channel resistance
- Best public cryptanalysis only gets 7/20 rounds
- Used in TLS, Linux's /dev/(u)random
- All around a better choice than RC4

# Block Ciphers in CTR Mode

CTR mode is a mode of operation where a block cipher is given a counter as plaintext, and the output is XOR'd with the actual plaintext to create the ciphertext.

- Makes behavior very similar to a stream cipher
- Inspired CBPRNGs
- Utilizes secure primitives, so security is asserted.



# Block Ciphers in CTR Mode: CTR\_DRBG

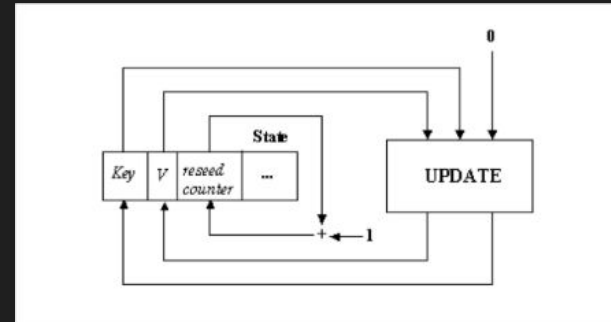
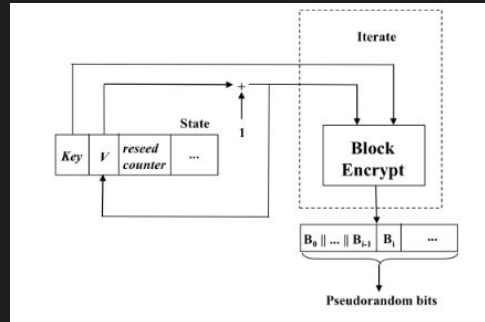
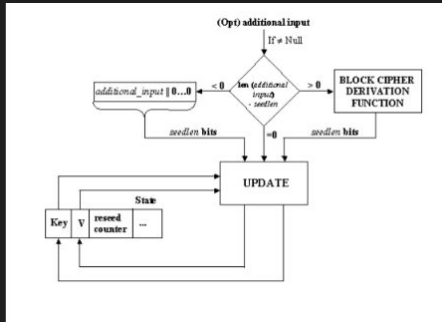
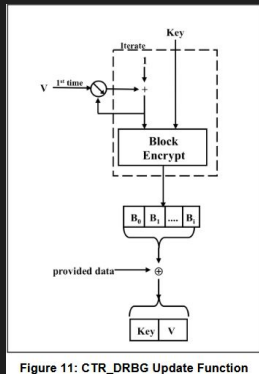
CTR\_DRBG is NIST's CTR Mode DRBG from the NIST SP-800-90A publication, officially supporting:

- AES-128/192/256
- TripleDES ("3DEA")

CTR\_DRBG can be most readily understood as a CTR Mode variant of the cipher with additional room for entropy injection, while also allowing an arbitrary bitstring length for output.

# Block Ciphers in CTR Mode: CTR\_DRBG Flaws

CTR\_DRBG, while practically secure, has an academic break in the case of requesting more bits than the block size. This is of primary concern with 3DES, but has not shown to be significant when AES is used as directed in terms of statistical properties or security. Schneier and Ferguson's *Fortuna* is often considered to be a superior example.





# Hash-Based CSPRNGs

Hash Based CSPRNGs utilize cryptographic hash functions to produce pseudo-random bits on output.

- Notable example: NIST Hash\_DRBG
- Particularly of interest given Sponge's indifferentiability and SHA3 standardization
- Traditionally used with SHA2/other MD hashes.

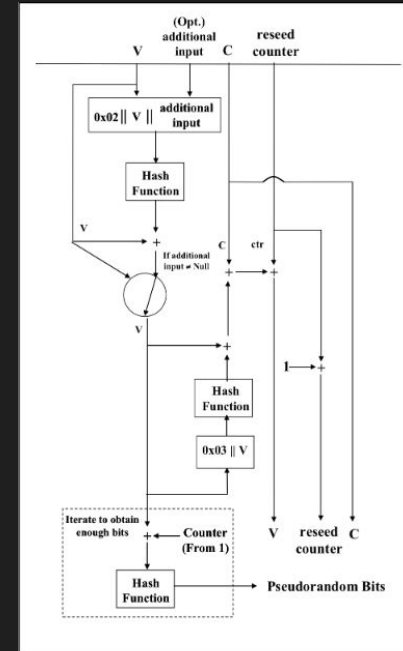
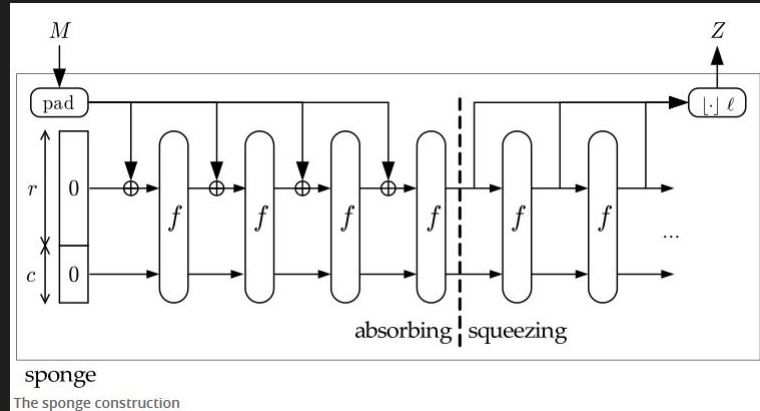


Figure 8: Hash\_DRBG

# Hash-Based CSPRNGs: Hash\_DRBG and SHA3

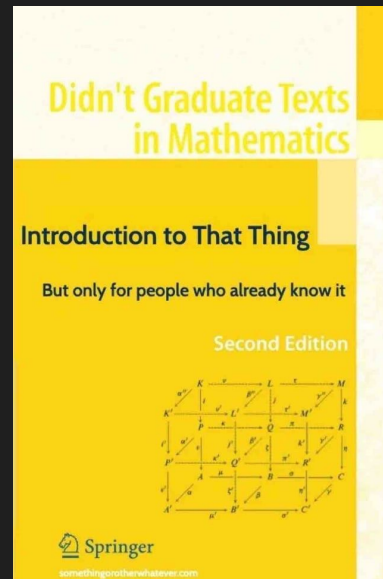
- Historically, while considered secure, Hash\_DRBG was not considered to be provably of good quality as a CSPRNG
- MD constructions not proven indifferentiable
- Sponge shown to be in both classical/quantum contexts
- Hash\_DRBG with SHA3/Keccak?



# Number-Theoretic Generators

Some generators make hardness assertions based on problems from number theory, similar to asymmetric cryptosystems.

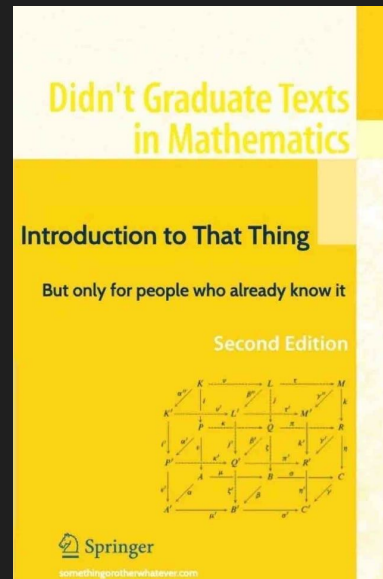
- Usually quite slow
- Not very popular
- Interesting nonetheless
- Examples:
  - Blum-Blum-Shub (QRP)
  - DUAL\_EC\_DRBG (ECDDH, XLog, TPP)
  - Blum-Micali (DLP)



# Number-Theoretic Generators

Some generators make hardness assertions based on problems from number theory, similar to asymmetric cryptosystems.

- Usually quite slow
- Not very popular
- Interesting nonetheless
- Examples:
  - Blum-Blum-Shub (QRP) ←
  - DUAL\_EC\_DRBG (ECDDH, XLog, TPP) ←
  - Blum-Micali (DLP)



# Blum-Blum-Shub

- Yes, that is this algorithm's real name
- Based on the hardness of the Quadratic Residuosity Problem (QRP)
- Defined as:

$$x_{n+1} = x_n^2 \bmod N$$

Where  $N$  is a large semiprime of factors  $p, q$  s.t.  
 $p \equiv q \equiv 3 \bmod 4$

- Typically, only a parity bit is output as a hardcore predicate

## Blum-Blum-Shub: The QRP

The QRP can be understood as follows.

Given a semiprime  $N$  with large prime factors  $p, q$ , consider the multiplicative group  $\mathbb{Z}_N^*$ . Half of the elements will not have  $r$  such that  $x \equiv r^2 \pmod{N}$ . The other half of the elements may have such an element, and determining if they do is computationally hard.

## Blum-Blum-Shub: The QRP

Knowing the QRP, we now relate it to BBS:

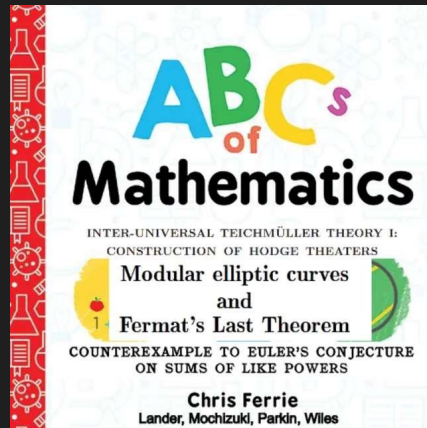
Given  $p \equiv q \equiv 3 \pmod{4}$ , any given quadratic residue will have 4 square roots, exactly one of which is also a quadratic residue. Given a quadratic residue  $x$ , we are tasked to find the parity bit of the root of  $x$  that is also a quadratic residue. As such, we must identify which root is a quadratic residue modulo  $N$ , which is hard.

At this point, the hardness can be shown to be equivalent to factoring.

# DUAL\_EC\_DRBG

DUAL\_EC\_DRBG was a proposed generator built on elliptic curve arithmetic.

- It was developed by the NSA
- It features a kleptographic backdoor
- This would have broken the entire cryptographic stack
- A fantastic lesson on the importance of nothing-up-my-sleeve numbers.





## DUAL\_EC\_DRBG

DUAL\_EC\_DRBG was defined as follows:

Let  $\mathcal{E}$  be an elliptic curve of the form  $y^2 = x^3 - 3x + b$  over a field of prime characteristic  $p$ .

Define two base points,  $P, Q \in \mathcal{E}$ .

Define the following functions:

$X(x, y) \rightarrow x$ : Extract the x-coordinate of a point in affine coordinates.

$t(x): \{0, 1\}^k \rightarrow \{0, 1\}^{k-n}$ : Truncates  $n$  MSB of data (default 16)

$g_p(x) = X(xP)$ : Perform scalar multiplication on  $P$ .

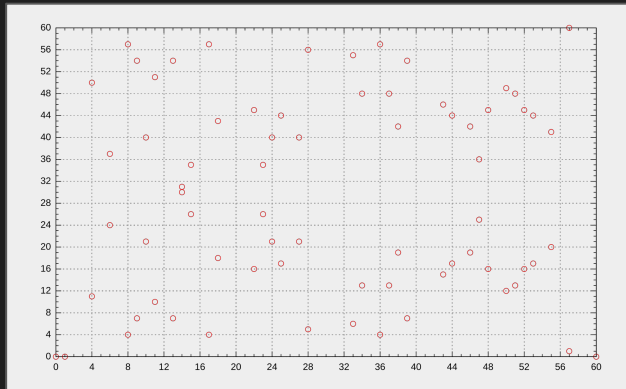
$g_q(x) = t(X(xQ))$ : Perform scalar multiplication on  $Q$ , extract the x-coordinate, and truncate it by  $n$  bits.

By standard, it additionally required usage of a NIST curve, 16-17 bit truncation, and you must use the standard  $P, Q$  for the curve chosen.

# DUAL\_EC\_DRBG: Function

Function of the generator is quite simple, and is as follows:

- Select a seed  $s \in \mathbb{F}_p$ .
- Perform  $V = g_p(s)$  to establish the first state.
- Output a value by calling  $g_q(V)$ .
- Transition state by calling  $g_p(V)$ .
- Repeat as required.



## DUAL\_EC\_DRBG: The Trick

- Since all NIST curves have a cofactor of 1, it's guaranteed that there is a scalar value  $s$  :  $sQ = P$ .
- Consider a single output,  $t$ , and the point it was derived from,  $A$ . There must exist some scalar  $r$  :  $A = rQ$ .
- $sA = srQ$  ( $A = rQ$ )
- $srQ = rP$  ( $sQ = P$ )
- $rP = \text{state}_2$

However,  $t$  is truncated, so finding  $A$  is hard, right?

## DUAL\_EC\_DRBG: The Trick

No! Since we only truncate 16 bits, it's as simple as guessing them and seeing if that x coordinate is valid on the curve equation. Once you do that, finding the y coordinate is as simple as running Tonelli-Shanks.

- Had this been in common use, the entire process of establishing a secure channel would have been broken.
- Unexplained P, Q are what allow it to work
- Figuring out the actual backdoor is still hard

## Sources

Can be found in original paper on GitHub at:

- <https://github.com/1nf0calypse/Cracking-Chaos/>
- Additionally contains implementations of every generator discussed here in Python3 with annotations

# Questions?

Contact Information:

Email: 1nfocalypse<at>protonmail<dot>com

Feel free to talk to me afterwards:

CPV 2-4:30 PM

Paper and Implementations:

<https://github.com/1nfocalypse/Cracking-Chaos/>