Pseudo Random Number Generator

ICMC USP SÃO CARLOS

GANESH

O que, e por que?

O que?

- PRNGs são funções que, a partir de um estado atual, geram um número que parece aleatório.

O que?

- PRNGs são funções que, a partir de um estado atual, geram um número que parece aleatório.
- Geralmente baseados em:
  - Recorrência linear
  - Matemática discreta
- Porém, novidades estão sendo testadas:
  - Teoria do caos
  - Curvas Elipticas

Por que (estudar)?

- Criptografia (especialmente criptografia de fluxo) depende muito de números pseudo aleatórios.
- Nem todos os PRNGs podem ser usados para criptografia, os que podem são chamados de CSPRNG (Cryptographically Secure)
- Explorar vulnerabilidades de PRNGs podem facilitar criptoanálise.

CSPRNG

- Um CSPRNG pode ser baseado nos mesmos conceitos em que baseamos criptografias:
  - Matemática discreta;
  - Teoria do Caos

- Mas também pode usar outras bases:
  - Criptografias;
  - Hashes

Exemplos de RNGs classicos

- LCG
- xorshift
- Mersenne Twister

Exemplos de CSPRNG comuns

- Blum-Micali
- Blum Blum Shub
- ChaCha20
- CryptGenRandom

Exemplos de CSPRNG novos

- K-Mapa logistico

- EC-OPRF (https://eprint.iacr.org/2017/111.pdf)

LCG

- Primeiro gerador criado.

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

- Mesmo quando era o único usado, sabia-se que haviam falhas, mas não sabiam como consertar.

- Útil para situações em que vários numeros sao necessarios, mas o sistema não é crítico

LCG

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

- Como aumentar o período:
  - M e C devem ser primos relativos (não compartilham nenhum fator primo)
  - a-1 deve ser divisível por todos os fatores primos de m.
  - se m for divisível por 4, a - 1 também deve ser divisível por 4.
- Para stdlib:
  - a = 11035234
  - c = 12345
  - m = 2^31
  - o resultado exclui o bit mais significativo

XORSHIFT

- 2 versões:
  - Com memória
  - Sem memória
- Sem memória, usa-se o estado atual, com bit shifts e xors.
- Com memória, mistura-se o mais novo com o mais antigo.

XORSHIFT sem memoria

```
X ^= X << 13
X ^= X >> 17
X ^= X << 5
```

32 bits

```
X ^= X << 13
X ^= X >> 7
   X ^= X << 17
```

64 bits

XORSHIFT com memoria

```
T ^= T << 11
T ^= T >> 8
T ^= S
T ^= S >> 19
```

Mersenne Twister

- Período muito maior que o LCG (2^(19937) - 1)
- Baseado em LFSR, com uma alteração (twist) envolvendo o primo de Mersenne igual ao período do gerador
- O estado é dado por um vetor de valores, que serão escolhidos um a um, e atualizados depois de n chamadas.

Mersenne Twister

- Se o valor pedido não excedeu o tamanho do estado, o i-ésimo valor do estado será processado, e retornado.

```
y := x ^ ((x >> u) & d)
y := y ^ ((y << s) & b)
y := y ^ ((y << t) & c)
return y^(y >> l)
```

- Sendo u,s,t,l,b,c,d parâmetros do gerador.

Mersenne Twister

- depois de n chamadas, é realizado o twist

$$x_{k+n} := x_{k+m} \oplus \left(({x_k}^u \mid\mid {x_{k+1}}^l) A \right) \qquad k = 0, 1, \ldots$$

$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots, a_0) \end{pmatrix} \qquad \boldsymbol{x}A = \begin{cases} \boldsymbol{x} \gg 1 & x_0 = 0 \\ (\boldsymbol{x} \gg 1) \oplus \boldsymbol{a} & x_0 = 1 \end{cases}$$

Mersenne Twister

- Para gerar o estado inicial a partir de um numero so, usamos a seguinte fórmula:

$$x_i = f \times (x_{i-1} \oplus (x_{i-1} >> (w-2))) + i$$

Blum Micali

- Esse gerador é baseado em matematica discreta, com uma fórmula muito parecida com a da criptografia RSA $x_{i+1} = g^{x_i} \mod p$
- Onde p é um primo grande, e g eh uma raiz primitiva de p. $x_i \leq \frac{p-1}{2}$
- Cada chamada do gerador devolve um bit. 1 se

Blum Blum Shub

- Também parecido com RSA, porém mais rápido de calcular

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

- Onde, assim como em RSA, M = pq, p e q sendo primos grandes

- Para evitar que o estado seja conhecido é retornado uma quantidade de bits menos

ChaCha20

- Usado no Linux, como PRNG padrão para coisas criptográficas
- Baseado na criptografia ChaCha20.
- O estado é dado por:
  - Uma constante de 128 bits;
  - Uma chave criptográfica de 256 bits (no caso de um PRNG, essa é a seed)
  - Um contador de 64 bits

ChaCha20

- Todos esses dados são separados na seguinte matriz:

Cons	Cons	Cons	Cons
Key	Key	Key	Key
Key	Key	Key	Key
Pos	Pos	Nonce	Nonce

- E então é realizado 10 “Double Rounds”, cada um com 4 “Quarter Rounds”.

ChaCha20

- Um quarter round é calculado como:

  a += b; d ^= a; d <<<= 16;

  c += d; b ^= c; b <<<= 12;

  a += b; d ^= a; d <<<=8;

  c += d; b ^= c; b <<<=7;

  Onde >>> e <<< são bitshifts com rotação;
- Um Double Round é calculado fazendo 4 quarter rounds, 1 para cada linha, depois mais 4 quarter rounds para as colunas;

CryptGenRandom

- Criado para todas as coisas criptográficas no windows
- Detalhes da implementação nunca foram revelados
  - MAS, um grupo de pesquisa em Israel conseguiu fazer reverse engineering e descobrir

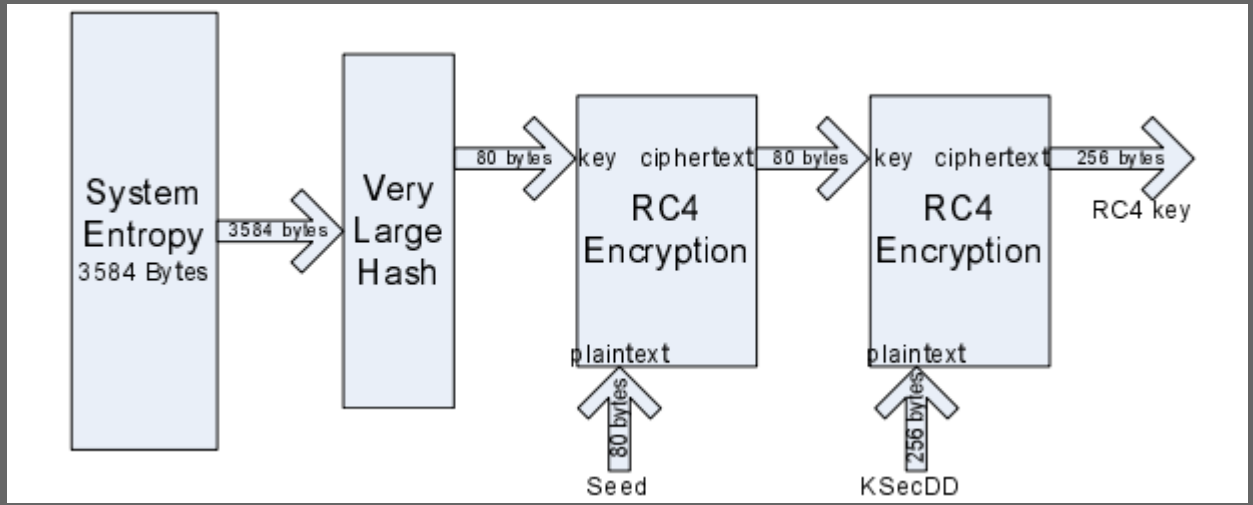
Fonte: https://eprint.iacr.org/2007/419.pdf

- A implementação obtida usa RC4 como base dos

CryptGenRandom

```
CryptGenRandom(Buffer, Len)
// output Len bytes to buffer
   while (Len>0) {
      R := R ⊕ get_next_20_rc4_bytes()
      State := State ⊕ R
      T := SHA-1'(State)
      Buffer := Buffer | T
        // | denotes concatenation
      R[0..4] := T[0..4]
        // copy 5 least significant bytes
      State := State + R + 1
      Len := Len − 20
   }
```

CryptGenRandom



K-Mapa Logistico

- Baseado em teoria do caos. Especificamente, no mapa logístico $x_{n+1} = r x_n \left(1 - x_n\right)$

- Com r ∈ (3.5, 4), X parece ser um valor aleatório ∈ (0,1).

- Apenas usar X nao eh bom o suficiente, então usou-se deep-zoom

K-Mapa Logistico

- Multiplica-se por 10^k, depois remove-se a parte inteira.
- Quanto mais fundo o “zoom”, maior a entropia (Conjectura)
- Caos eh muito sensivel a qualquer variação, então é recomendado usar um k bem menor que a precisão da variável usada.

EC-OPRF

- Baseado em um Hash (H), um sal “desconhecido” (S), uma máscara (M), e (mais importante) uma curva elíptica [E(a,b,p)].
- Esse gerador é, na verdade, um protocolo para pedir que um servidor semi-confiável retorne um número pseudo-aleatório baseado em uma string
- Referencia: https://eprint.iacr.org/2017/111.pdf

EC-OPRF

- O paper descreve o processo em 5 partes:
  - O cliente opera um Hash *seguro* sobre a string de entrada
    - Como exemplo no paper, usa-se SHA-256
  - A Hash é mapeada em uma curva elíptica, para um ponto P
  - O cliente gera um mascara pseudo aleatória (Cuja função não precisa ser tão segura)
  - O ponto mascarado é enviado para um servidor, que o salga
  - O cliente recebe a resposta do servidor e remove a máscara dos dados.

EC-OPRF

- Todos os passos são realizados *módulo p*;
- Como estamos trabalhando com aritmética modular, a máscara pode ser uma simples multiplicação *módulo* p;
- Portanto, retirar a máscara é simplesmente multiplicar pelo inverso de M *módulo* p;

EC-OPRF

- O throughput é muito baixo para usar como RNG, mas os dados são difíceis de obter
- Dessa forma, podemos usar esse PRNG para gerar o estado inicial de um PRNG
- Ou para gerar as chaves de alguma criptografia