Computer Science and Algorithmics, PWr

Mirosław Kutyłowski

PRNG

PRNG - pseudorandom number generator

– input is a random seed s

– $\mathrm{PRNG}(s)$ is a long string that “looks as random”

– PRNG is a deterministic function

Application: Stream ciphers (szyfry strumieniowe)

key: k (chosen at random)

encrypting a long message M:

– $\kappa := \mathrm{PRNG}(k)$ truncated to the length of M

– $c := \kappa \otimes M \quad (\mathrm{bitwise\, XOR})$

decryption:

– $M := c \otimes \kappa$

Protection against reuse of κ:

– the PRNG has an internal state

– state updated after each use

Desired properties of PRNG

– $\mathrm{PRNG}(k)$ truncated to m bits is a uniformly distributed random variable if k is uniformly distributed seed

$\rightarrow$ **impossible** if $m > \mathrm{length}(k)$:

there are 2^m bitstrings of length m

but there are only $2^{\mathrm{length}(k)}$ outputs of PRNG of length m

Computational indistinguishability

it is enough if you cannot distinguish an output of PRNG from a random source

Left-or-right game:

in a blackbox: either

i. a real random source with uniform distribution, or

ii. a PRNG initiated with a random seed

Task: guess what is in the blackbox while observing its output

Advantage ϵ

if probability to win is $\frac{1}{2} + \epsilon$

Formal definition of computational indistiguishability

two sources X_0, X_1 are computationally indistinguishable if for any (polynomial) algorithm A:

$$\left| \quad \Pr(A(x) = 0 | x = X_1) \quad - \quad \Pr(A(x) = 0 | x = X_0) \quad \right|$$

is negligible

Security analysis for proposed PRNG

trying to find an A where it is not true, where A is (to some degree) practical

called "***distinguishing attack***"

Particular formulation

algorithm A should guess whether the output comes from X_0 or X_1

Theorem

Both formulations are equivalent

consequences:

- 1st formulation e**xcludes any observable difference** of X_0 and X_1
- 2nd formulation: **easier for checking** properties of PRNG

Unpredictability

given the output $b_0 \ldots b_n$ of an PRNG it is infeasible to predict b_{n+1}

Backwards security

if internal state s_m is leaked, then it is infeasible to derive any information on $b_0 b_1 \ldots b_{m-1}$

Applications:

think about confidentiality of phone conversations (they are secured with stream ciphers!)

PRNG

- deterministic function D from n bit seeds to m bit strings
- output of D computationally indistinguishable from a real random source with uniform distribution over m-bit strings

Construction with HCP (hardcore predicate) , $m = 1$:

- f is a one-way permutation, h – hardcore predicate
- $G(s) := f(s)||h(s)$

(a toy example as the output is longer than the seed by 1 bit only)

Example

one way function: $f(x) = x^d \mod n$ (RSA ciphertext)

$h(f(x)) = x \mod 2$

More efficient construction

- $s_0 := s$
- $s_1 || b_0 := G(s_0)$
- $s_2 || b_1 := G(s_1)$
- $\ldots$
- $s_{n+1} || b_n := G(s_n)$
- output $b_0 b_1 \ldots b_n$

Thm. If G is constructed via one-way permutation f and and hardcore predicate h, then the above construction yields a PRNG indistinguishable from real random source

Draft of a proof

Version i

- $s_0 := s$
- b_0 at random
- b_1 at random
- $\dots$
- s_i at random, b_{i-1} at random
- $s_{i+1} || b_i := G(s_i)$
- $\dots$
- $s_{n+1} || b_n := G(s_n)$
- output $b_0 b_1 \dots b_n$

Distinguishing version i and version $i+1 \Rightarrow$ distinguisher breaking HCP

Draft continued

version 0 — PRNG

version 1

version 2

...

version n — real random source

PRNG practice

- **initialization:** the seed recomputed with auxiliary input from the user and internal randomness "*entropy*"
- **reseeding:** similar as above, entropy bits taken

goals:

- internal state of PRNG is user dependent (but dependance limited)
- limited number of bits with one deterministic function
- noise from entropy

typical operation cycle:

- after initialize/refresh: run for some time, discarding output
- work and yield output
- refresh when the `refresh_conter` reaches 0

RC4

PRNG based on the idea of a random shuffling of cards

Ron's Cipher – designed by Ronald Rivest from MIT

some weaknesses

phases:

1. **initialization** with the secret key, no output, runs for some time
2. **generation of random bytes** : internal state changed at every stage

RC4, initialization phase

```
for i from 0 to 255
    S[i] := i
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod  keylength]) mod 256
    swap(S[i],S[j])
```

RC4 output generation

```
i:= 0
j:= 0

while output is needed:
    i:= (i + 1) mod 256
    j:= (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
```

ChaCha

- European algorithm, former version: Salsa, from eStream competition
- design goals: easy software implementation, any platform, ...
- follows architecture borrowed from stream ciphers
- working on 32 bit words

quarter-round of ChaCha20

1. $a = a + b$; $d = d \operatorname{xor} a$; $d = d \lll 16$
2. $c = c + d$; $b = b \operatorname{xor} c$; $b = b \lll 12$
3. $a = a + b$; $d = d \operatorname{xor} a$; $d = d \lll 8$
4. $c = c + d$; $b = b \operatorname{xor} c$; $b = b \lll 7$

Initialization

Chacha matrix 4x4: (where 'input' = ‘block counter'+nonce)

const	const	const	const
key	key	key	key
key	key	key	key
input	input	input	input

Output generation

20 rounds executed, the contents of the matrix is the output,

round: consists of 8 quarter-rounds

- quarter-rounds on: 1st column, 2nd column, 3rd column, 4th column
- quarter-round on diagonals

$$\mathrm{quarter-round}(x0, x5, x10, x15),$$

$$\mathrm{quarter-round}(x1, x6, x11, x12)$$

$$\mathrm{quarter-round}(x2, x7, x8, x13)$$

$$\mathrm{quarter-round}(x3, x4, x9, x14)$$

output: the matrix contents

restart: with the next block counter

Open competitions for cryptographic primitives

- mainly by NIST
- open evaluation
- the whole community involved in looking for weaknesses
- works better than design in secrecy (e.g. GHOST)

Alternative constructions

- from block encryption
- from asymmetric algorithms
- from hash functions

Hardware generators:

- physical source might be ok but:
  - $\longrightarrow$ measurement bias
  - $\longrightarrow$ digital processing
  - $\longrightarrow$ physical attacks (low temperature, . . .)
  - $\longrightarrow$ non-verifiability
- quantum generators: expensive, poor output
- NIST recommendation some time ago: use PRNG

Remembering long random strings

- many protocols need it
- instead of storing the output of PRNG, just remember the seed and reconstruct
- ChaCha - easy (generate again with the same key and block number)
- basic PRNG wihout restarting: not efficient
- option: tree-like construction (in the textbook)