- Agenda: explore **(pseudo-)random bit generation**, via
  1. an "in theory", i.e., design-oriented perspective, and
  2. an "in practice", i.e., implementation-oriented perspective.
- Caveat!

$$\sim 2 \text{ hours} \Rightarrow \text{introductory, and (very) selective (versus definitive) coverage.}$$

- Bad news: in *theory*, we need to consider each of
  1. random *bit*, i.e., an
  $$x \in \{0,1\}$$
  which is random,
  2. random *bit sequence*, i.e., an
  $$x \in \{0,1\}^n$$
  which is random (e.g., for an AES cipher key $k$),
  3. random *number*, i.e., an
  $$x \in \{0,1,\ldots,n-1\}$$
  which is random (e.g., for an RSA modulus $N = p \cdot q$).

- Good news: in *practice*, we don't because
  - 1. $\Rightarrow$ 2.
    - concatenate $n$ random bits together, i.e.,

$$x = x_0 \parallel x_1 \parallel \cdots \parallel x_{n-1},$$

    - produce $x$ as output.
  - 2. $\Rightarrow$ 3.
    - if $n = 2^{n'}$ for some integer $n'$, then
      - generate an $n'$-bit sequence $x'$ per the above,
      - interpret $x'$ as the integer

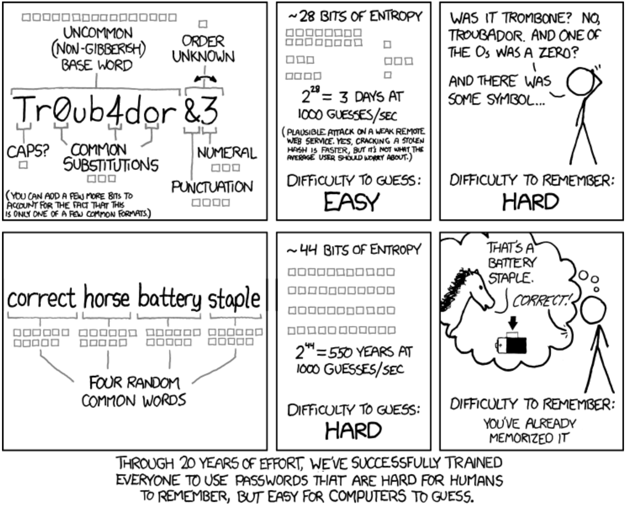
$$x = \sum_{i=0}^{i<n'} x'_i,$$

      - produce $x$ as output.
    - if $n \neq 2^{n'}$ for any integer $n'$, then
      - let $n'$ be the smallest integer such that $2^{n'} > n$,
      - generate an $n'$-bit sequence $x'$ per the above,
      - interpret $x'$ as the integer

$$x = \sum_{i=0}^{i<n'} x'_i,$$

      - if $x \geq n$, reject (or discard) it and try again; otherwise, if $x < n$, produce $x$ as output.

$\therefore$ we can focus on random bits (and ignore numbers).




---

https://xkcd.com/936

### Definition

The concept of **entropy** is a measure of uncertainty with respect to a random variable. Less formally, the entropy of some $x$ relates to how much you know (resp. do not know) about $x$: if some $x$ could be one of $2^n$ possible values, it is said to have $n$ bits of entropy. In addition, we say

1. an $x$ with $n > 0$ bits of entropy is termed **entropic**, and
2. if an entropic $x$ has negligible probability of having been generated before, it is deemed **fresh entropy**.

### Definition

The concept of **entropy** is a measure of uncertainty with respect to a random variable. Less formally, the entropy of some $x$ relates to how much you know (resp. do not know) about $x$: if some $x$ could be one of $2^n$ possible values, it is said to have $n$ bits of entropy. In addition, we say

1. an $x$ with $n > 0$ bits of entropy is termed **entropic**, and
2. if an entropic $x$ has negligible probability of having been generated before, it is deemed **fresh entropy**.

- Example: given a 32-bit sequence $x$,
  - if $x$ is random, then it has 32 bits of entropy,
  - if $x_0 = 0$ and $x_1 = 1$ (i.e., the two LSBs of $x$ are known), then it has 30 bits of entropy,
  - if $\mathrm{HW}(x) = 14$ (i.e., $x$ has Hamming weight 14), then it has $\sim 29$ bits of entropy.

## Definition

A **noise source** is a non-deterministic, physical process which provides a means of generating an *unconditioned* (or raw) entropic output.

## Definition

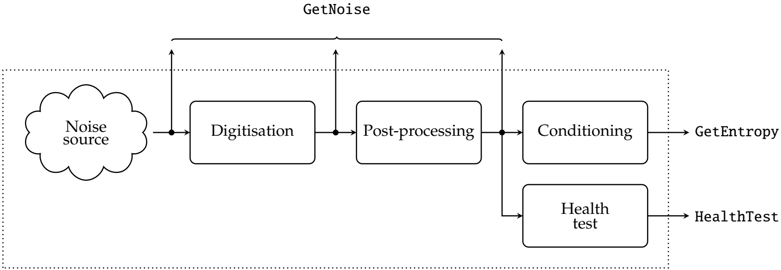
A **noise source** is a non-deterministic, physical process which provides a means of generating an *unconditioned* (or raw) entropic output.

- Example (see [8, Section 5.2], or [14, Section 3]):
  1. hardware-based:
     - time between emission of (e.g., $\alpha$ or $\beta$) particles during radioactive decay,
     - thermal (or Johnson-Nyquist) noise stemming from a resistor or capacitor,
     - frequency instability (or "jitter") of a ring oscillator,
     - fluctuation of hard disk seek-time and access latency,
     - noise resulting from a disconnected audio input (or ADC),
     - ...
  2. software-based:
     - a high resolution system clock or cycle counter,
     - elapsed time between user input (e.g., key-presses or mouse movement),
     - content of input/output buffers (e.g., disk caches),
     - operating system state (e.g., load) or events (e.g., network activity),
     - ...

## Definition

An **entropy source** is a construction, based on a noise source, which provides a means of generating a *conditioned* entropic output.

## Model [16, Section 2.2]



## Definition

Per [15, Section 4], an **ideal random bit-sequence**

$$x = \langle x_0, x_1, \ldots x_{n-1} \rangle$$

will exhibit the following properties

| | | | |
|---|---|---|---|
| 1. | **unpredictable** | $\Rightarrow$ | the probability of guessing $x_i$ is close to $\frac{1}{2}$ |
| 2. | **unbiased** | $\Rightarrow$ | $x_i = 0$ and $x_i = 1$ occur with equal probability |
| 3. | **uncorrelated** | $\Rightarrow$ | $x_i$ and $x_j$ are statistically independent |

and contain $n$ bits of entropy.

## Definition

Per [15, Section 4], a **pseudo-random bit-sequence**

$$x = \langle x_0, x_1, \ldots x_{n-1} \rangle$$

"looks random", i.e., exhibits the same properties as an ideal random sequence, *but* is generated algorithmically and thus likely contains less than $n$ bits of entropy.

## Definition

A **Random Bit Generator (RBG)** can be used to generates a sequence of random bits. There are two more specific cases, namely

| | | |
|---|---|---|
| **True Random Bit Generator (TRBG)** | ≡ | **Non-deterministic Random Bit Generator (NRBG)** |
| **Pseudo-Random Bit Generator (PRBG)** | ≡ | **Deterministic Random Bit Generator (DRBG)** |

with the right-hand terms preferred by [15]. Based on this, it is reasonable to say that

$$\text{TRBG} \equiv \text{NRBG} \simeq \text{entropy source.}$$

## Definition

A **Random Bit Generator (RBG)** can be used to generates a sequence of random bits. There are two more specific cases, namely

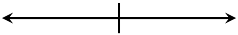
| | | |
|---|---|---|
| **True Random Bit Generator (TRBG)** | $\equiv$ | **Non-deterministic Random Bit Generator (NRBG)** |
| **Pseudo-Random Bit Generator (PRBG)** | $\equiv$ | **Deterministic Random Bit Generator (DRBG)** |

with the right-hand terms preferred by [15]. Based on this, it is reasonable to say that

$$\text{TRBG} \equiv \text{NRBG} \simeq \text{entropy source.}$$

- Idea: informally at least,

**TRBG**
- unpredictable,
- hardware-based,
- may have high latency,
- can be high quality.



**PRBG**
- predictable,
- software-based,
- can have low latency,
- may be low quality.

$\therefore$ we'll consider a *hybrid* construction.

## Definition

Consider a deterministic, polynomial-time algorithm $G$. Given a **seed** $\varsigma \in \{0,1\}^{n_\varsigma}$ as input, it produces $G(\varsigma) \in \{0,1\}^{n_r}$ as output where $n_r = f(n_\varsigma)$ for some polynomial function $f$. As such, we call $G$ a **Pseudo-Random Generator (PRG)** if

1. for every $n_\varsigma$ it holds that $n_r > n_\varsigma$, and
2. for all polynomial-time destinguishers $D$, there exists a negligible function $\mathsf{negl}$ such that

   $$|\ \Pr[D(G(\varsigma)) = 1]\ -\ \Pr[D(r) = 1]\ |\ \leq\ \mathsf{negl}(n_\varsigma)$$

   where $\varsigma$ and $r$ are chosen uniformly at random from $\{0,1\}^{n_\varsigma}$ and $\{0,1\}^{n_r}$ respectively.

## Syntax

Having fixed the (finite) space $\mathcal{S}$ of states, a concrete **Pseudo-Random Generator (PRG)** is defined by

1. an algorithm $\textsc{Seed} : \mathbb{Z} \times \{0,1\}^{n_\varsigma} \rightarrow \mathcal{S}$ that
   - accepts a security parameter and an $n_\varsigma$-bit seed as input, and
   - produces an initial state as output
2. an algorithm $\textsc{Update} : \mathcal{S} \rightarrow \mathcal{S} \times \{0,1\}^{n_b}$ that
   - accepts a current state as input, and
   - produces a next state and an $n_b$-bit block of pseudo-random bits as output.

- **Translation**: assuming $n_r = l \cdot n_b$ for some $l$, then

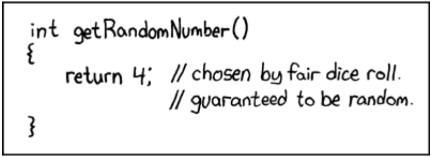
  1. use TRBG $\rightsquigarrow$ { generate a sufficiently large, high-entropy seed $\varsigma$
  2. use PRBG $\rightsquigarrow$

$$\left\{ \begin{array}{lllll} \theta[0] & & \leftarrow & \textsc{Seed}(\lambda, \varsigma) \\ \theta[1] & , b[0] & \leftarrow & \textsc{Update}(\theta[0]) \\ \theta[2] & , b[1] & \leftarrow & \textsc{Update}(\theta[1]) \\ & & \vdots & \\ \theta[i+1] & , b[i] & \leftarrow & \textsc{Update}(\theta[i]) \\ & & \vdots & \end{array} \right.$$

  meaning that

$$b = \underbrace{\underbrace{b[0]}_{n_b\text{-bits}} \parallel \underbrace{b[1]}_{n_b\text{-bits}} \parallel \cdots \parallel \underbrace{b[l-1]}_{n_b\text{-bits}}}_{l \cdot n_b = n_r\text{-bits}} \equiv G(\varsigma)$$

  provides the output required per the PRG definition.




---
https://xkcd.com/221

- **Problem**: we need to assess the quality of our construction (and output from it).
- **Solution**:
  1. for *some* instanciations, we can develop a proof,
  2. for *some* instanciations, we must apply
     - online (e.g., continuously or periodically *during* use), and/or
     - offline (i.e., once *before* use)

     statistical tests (see, e.g., [8, Section 5.4]) to sample outputs; note that
     - the intention is to detect weakness (meaning a PRBG can only be rejected by a test),
     - the conclusion is itself probabilistic, meaning use of multiple tests amplifies confidence.

## Definition

A PRBG is said to pass all **statistical tests** iff. no polynomial-time algorithm can, with probability greater than $\frac{1}{2}$, distinguish the output from a ideal random bit-sequence of the same length.

## Definition

A PRBG is said to pass the **next-bit test** iff. no polynomial-time algorithm can, with probability greater than $\frac{1}{2}$, predict the $(n+1)$-th bit of output given the previous $n$ bits.

## Theorem (Yao [11])

If a PRBG passes the next-bit test, it will pass all statistical tests.

## Definition

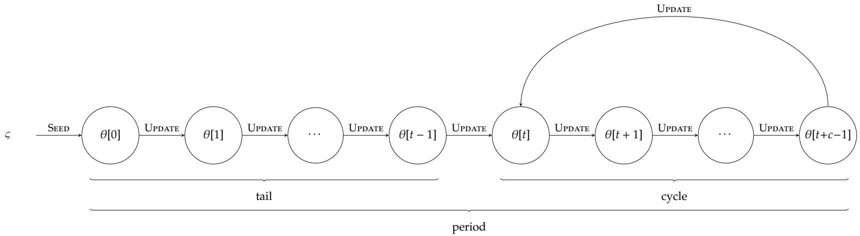
Per [15, Section 4], imagine an attacker compromises the PRBG state at time $t$: we term a PRBG **back-tracking resistant** (resp. **prediction resistant**) if said attacker cannot distinguish between an (unseen) PRBG output at time $t' < t$ (resp. $t' > t$) and an ideal random bit-sequence of the same length.

## Definition

A **Cryptographically Secure Pseudo-Random Bit Generator (CS-PRBG)** is simple a PRBG whose properties make it suitable for use within a cryptographic use-case. A CS-PRBG should (at least)

1. be a PRBG of sufficient quality, i.e., pass the next-bit test, and
2. resist state compromise attacks, i.e., be back-tracking and prediction resistant.

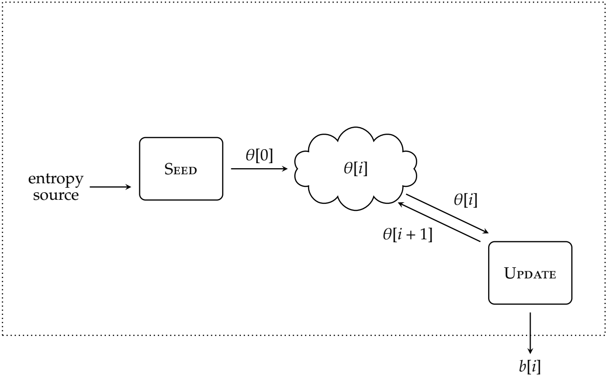
- Problem: our construction is deterministic, so
  - the same $\varsigma$ will yield the same $\theta[0]$ and hence any $\theta[j]$ for $j > 0$,
  - recovery of $\varsigma$ allows computation of any $\theta[j]$ for $j \geq 0$,
  - recovery of $\theta[i]$ allows computation of any $\theta[j]$ for $j > i$,
  - the set $\mathcal{S}$ is finite, so per



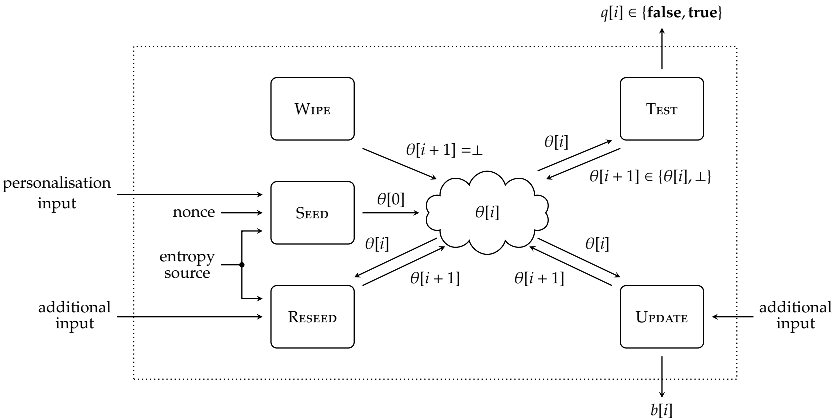

  the state, and thus also the output, will eventually cycle.

- Solution:
  1. select parameters that mitigate such issues, and
  2. introduce selected *non*-determinism.

## Definition ([15, Figure 1])



## Definition ([15, Figure 1])



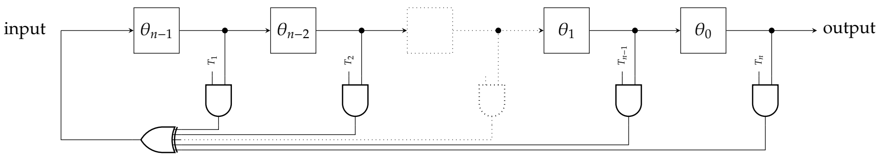
# Part 2: in practice (1)

- (Sub-)agenda: explain selected, example designs, organised into 4 classes, i.e.,
  1. "classic",
  2. software-oriented,
  3. hardware-oriented,
  4. system-oriented,

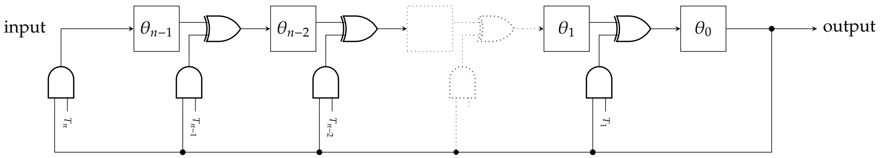
  with a focus on design properties and trade-offs between them, e.g.,
  - efficiency,
  - security, i.e., quality of (pseudo-)random output,
  - interface,
  - assumptions,
  - ...

- Design: **Linear-Feedback Shift Registers (LFSR)** [5, 6].

## Algorithm (type-1, or "external" or Fibonacci, LFSR)




## Algorithm (type-2, or "internal" or Galois, LFSR)



- Design: **Blum-Blum-Shub (BBS)** [10].

## Algorithm (BBS.Seed)

**Input:** A security parameter $\lambda$, and a seed $\varsigma$
**Output:** An initial state $\theta[0]$

Use entropy provided by $\varsigma$ to perform the following steps:

1. Select two random $(\lambda/2)$-bit primes $p$ and $q$ such that $p \equiv q \equiv 3 \pmod 4$, and compute $N = p \cdot q$.
2. Select a random $s \in \{0, 1, \ldots, N - 1\}$ such that $\gcd(s, N) = 1$.
3. Compute $s[0] = s^2 \pmod N$.
4. Return $\theta[0] = (N, s[0])$.

- Design: **Blum-Blum-Shub (BBS)** [10].

## Algorithm (BBS.Update)

**Input:** A current state $\theta[i] = (N, s[i])$
**Output:** A next state $\theta[i+1]$, and $n_b = 1$ bit pseudo-random output $b[i]$

1. Compute $s[i+1] = s[i]^2 \pmod{N}$.
2. Let $b[i] = s[i+1] \pmod{2}$, i.e., $b[i] = \text{LSB}(s[i+1])$.
3. Return $\theta[i+1] = (N, s[i+1])$ and $b[i]$.

- Design: **ANSI X9.31** [13, Appendix A.2.4].

## Algorithm (X9.31.Seed)

**Input:** A security parameter $\lambda$, and a seed $\varsigma$
**Output:** An initial state $\theta[0]$

1. Use $\lambda$ to select a block cipher with an $n_k$-bit key size and $n_b$-bit block size, e.g.,

| | | | |
|---|---|---|---|
| 3DES | $\rightsquigarrow$ | $n_b = 64$, | $n_k = 192$ |
| AES-128 | $\rightsquigarrow$ | $n_b = 128$, | $n_k = 128$ |
| AES-192 | $\rightsquigarrow$ | $n_b = 128$, | $n_k = 192$ |
| AES-256 | $\rightsquigarrow$ | $n_b = 128$, | $n_k = 256$ |

2. Use entropy provided by $\varsigma$ to derive an $n_k$-bit cipher key $k$ (or pre-select a $k$ for the PRBG).
3. Use entropy provided by $\varsigma$ to derive an $n_b$-bit block $s[0]$.
4. Return $\theta[0] = (k, s[0])$.

- Design: **ANSI X9.31** [13, Appendix A.2.4].

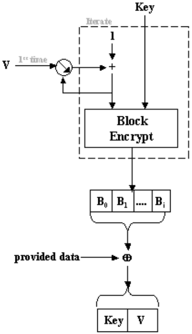
## Algorithm (X9.31.Update)

**Input:** A current state $\theta[i] = (k, s[i])$
**Output:** A next state $\theta[i+1]$, and $n_b$-bit pseudo-random output $b[i]$

1. Compute $t' = \textsc{Enc}(k, t)$, where $t$ is a $n_b$-bit representation of the current time.
2. Compute $b[i] = \textsc{Enc}(k, t' \oplus s[i])$.
3. Compute $s[i+1] = \textsc{Enc}(k, t' \oplus b[i])$.
4. Return $\theta[i+1] = (k, s[i+1])$ and $b[i]$.

- Design: **NIST CTR_DRBG** [15, Section 10.2.1].

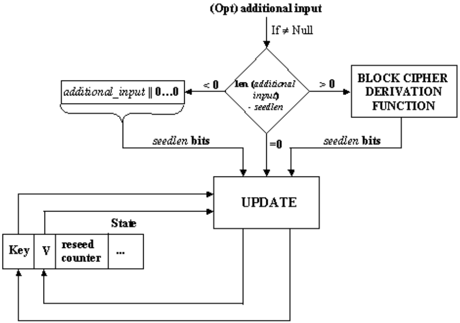
## Algorithm (CTR_DRBG.Update )




https://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

- Design: **NIST CTR_DRBG** [15, Section 10.2.1].

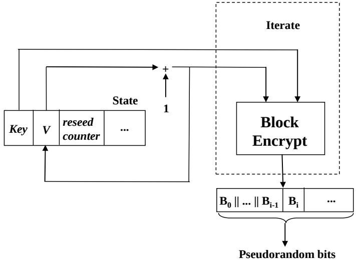
**Algorithm (CTR_DRBG.INSTANTIATE)**




https://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

- Design: **NIST CTR_DRBG** [15, Section 10.2.1].

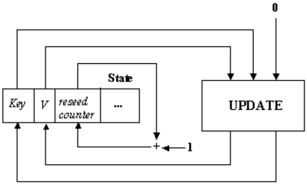
**Algorithm (CTR_DRBG.Generate )**




https://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

- Design: **NIST CTR_DRBG** [15, Section 10.2.1].

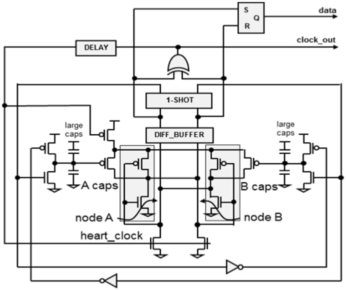
Algorithm (CTR_DRBG.Reseed)




https://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

- Design: **Intel Secure Key** [12].

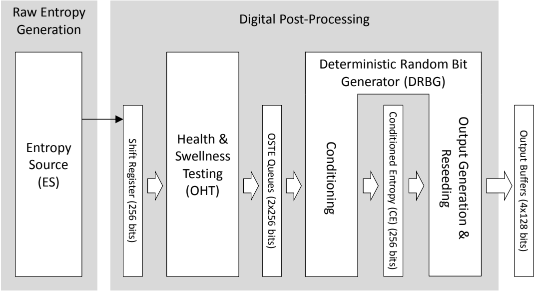
## Algorithm (RdRand entropy source)




https://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf

- Design: **Intel Secure Key** [12].

## Algorithm (RdRand TRBG)




https://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf

# Part 2: in practice (7)

Class #3: hardware-oriented

- Design: **Intel Secure Key** [12].

## Listing (RdRand interface)

```
bool rdrand64( uint64_t* r ) {
  bool success;

  asm( "rdrand %0 ; setc %1"
       : "=r" (*r), "=qm" (success) );

  return success;
}
```

## Listing (RdRand interface)

```
bool rdrand64_retry( uint64_t* r, int l ) {
  int i = 0;

  do {
    if( rdrand64( r ) ) {
      return true;
    }
  } while( i++ < l );

  return false;
}
```

- Design: **Linux**.
  - circa 1994(ish):
    - maintain entropy pool $\theta[i]$, injecting entropy, e.g., from system-related events,
    - define a predicate

$$P(\theta[i]) = \begin{cases} \textbf{false} & \text{if estimated entropy in } \theta[i] \text{ is deemed insufficient} \\ \textbf{true} & \text{if estimated entropy in } \theta[i] \text{ is deemed sufficient} \end{cases}$$

      based on the concept of entropy estimation,
    - expose $\theta[i]$ to user-space via the (pseudo) files

$$\text{write to } \texttt{/dev/random} \simeq \text{inject entropy into } \theta[i]$$

$$\text{read from } \texttt{/dev/random} \simeq \begin{cases} \text{if } P(\theta[i]) = \textbf{false}\text{, block then sample from PRNG (re)seeded from } \theta[i] \\ \text{if } P(\theta[i]) = \textbf{true}\text{, then sample from PRNG (re)seeded from } \theta[i] \end{cases}$$

$$\text{read from } \texttt{/dev/urandom} \simeq \text{sample from PRNG (re)seeded from } \theta[i]$$

- Design: **Linux**.
  - circa 2014(ish):
    - update re. additional system call

```
ssize_t getrandom( void* x, size_t n, unsigned int flags )
```

where

$$\texttt{getrandom} \simeq \begin{cases} \text{if PRNG has not been initialised, then do block} \\ \text{if PRNG has been initialised, then do not block} \end{cases}$$

    - this yields clear(er) semantics, and avoids need for file handle.

- Design: **Linux**.
  - circa 2016(ish):
    - update re. PRNG, which is changed from being based on SHA-1 to ChaCha20,
    - this yields, e.g., lower latency with respect to sampling output.

- Design: **Linux**.
  - circa 2020(ish):
    - update re. file-based semantics

$$\texttt{/dev/urandom} \simeq \text{do not block}$$

$$\texttt{/dev/random} \simeq \begin{cases} \text{if PRNG has not been initialised, then do block} \\ \text{if PRNG has been initialised, then do not block} \end{cases}$$

# Conclusions

### Quote

*Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.*

– *von Neumann* (https://en.wikiquote.org/wiki/Randomness)

### Quote

*The generation of random numbers is too important to be left to chance.*

– *Coveyou* (https://en.wikiquote.org/wiki/Randomness)

### Quote

*The design of such pseudo-random number generation algorithms, like the design of symmetric encryption algorithms, is not a task for amateurs.*

– *Eastlake, Schiller, and Crocker* [14]

# Conclusions

- Take away points:
  1. A high-quality source of randomness is fundamental to more or less *every* security proof: it might be an assumption in in theory, but in practice this issue requires care.
  2. Iff. you need to develop your own PRBG implementation, use a standard (e.g., NIST SP800-90A [15]) design or framework ...
  3. ... often such a design can leverage a primitive (e.g., a block cipher) you need anyway, thus reducing effort, attack surface, etc.
  4. Some golden rules:
     - use a large, high-entropy seed,
     - avoid reliance on a single entropy source where possible,
     - opt for a cryptographically secure design and ensure it is parameterised correctly,
     - hedge against failure via robust pre- and post-processing where need be,
     - include quality tests on pseudo-randomness generation (e.g., alongside functional unit testing),
     - don't compromise security for efficiency,
     - ...

# Additional Reading


- *Wikipedia: Randomness*. URL: https://en.wikipedia.org/wiki/Randomness.
- *Wikipedia: Pseudorandomness*. URL: https://en.wikipedia.org/wiki/Pseudorandomness.
- *Wikipedia: /dev/random*. URL: https://en.wikipedia.org/wiki/dev/random.
- *Wikipedia: RDRAND*. URL: https://en.wikipedia.org/wiki/RDRAND.
- K.H. Rosen. "Chapter 7: Discrete probability". In: *Discrete Mathematics and Its Applications*. 7th ed. McGraw Hill, 2013.
- A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. "Chapter 5: Pseudorandom bits and sequences". In: *Handbook of Applied Cryptography*. CRC, 1996. URL: http://cacr.uwaterloo.ca/hac/about/chap5.pdf.
- D. Johnston. *Random Number Generators – Principles and Practices: A Guide for Engineers and Programmers*. 1st ed. De | G Press, 2018.
- D. Eastlake, J. Schiller, and S. Crocker. *Randomness Requirements for Security*. Internet Engineering Task Force (IETF) Request for Comments (RFC) 4086. 2005. URL: http://tools.ietf.org/html/rfc4086.

# References


[1] *Wikipedia: /dev/random*. URL: https://en.wikipedia.org/wiki/dev/random (see p. 43).

[2] *Wikipedia: Pseudorandomness*. URL: https://en.wikipedia.org/wiki/Pseudorandomness (see p. 43).

[3] *Wikipedia: Randomness*. URL: https://en.wikipedia.org/wiki/Randomness (see p. 43).

[4] *Wikipedia: RDRAND*. URL: https://en.wikipedia.org/wiki/RDRAND (see p. 43).

[5] S.W. Golomb. *Shift Register Sequences*. 3rd ed. https://doi.org/10.1142/9361. Aegean Park Press, 2017 (see p. 25).

[6] M. Goresky and A. Klapper. *Algebraic Shift Register Sequences*. 1st ed. https://doi.org/10.1017/CBO9781139057448. Cambridge University Press, 2012 (see p. 25).

[7] D. Johnston. *Random Number Generators – Principles and Practices: A Guide for Engineers and Programmers*. 1st ed. De | G Press, 2018 (see p. 43).

[8] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. "Chapter 5: Pseudorandom bits and sequences". In: *Handbook of Applied Cryptography*. CRC, 1996. URL: http://cacr.uwaterloo.ca/hac/about/chap5.pdf (see pp. 7, 8, 18, 43).

[9] K.H. Rosen. "Chapter 7: Discrete probability". In: *Discrete Mathematics and Its Applications*. 7th ed. McGraw Hill, 2013 (see p. 43).

[10] L. Blum, M. Blum, and M. Shub. "A Simple Unpredictable Pseudo-Random Number Generator". In: *SIAM Journal on Computing* 15.2 (1986), pp. 364–383 (see pp. 26, 27).

[11] A.C. Yao. "Theory and application of trapdoor functions". In: *Symposium on Foundations of Computer Science (SFCS)*. 1982, pp. 80–91 (see p. 19).

[12] *Intel Digital Random Number Generator (DRNG) – Software Implementation Guide*. Tech. rep. Intel Corp., 2012. URL: http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf (see pp. 34–36).

[13] *Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry*. American National Standards Institute (ANSI) Standard X9.31. 1993 (see pp. 28, 29).

[14] D. Eastlake, J. Schiller, and S. Crocker. *Randomness Requirements for Security*. Internet Engineering Task Force (IETF) Request for Comments (RFC) 4086. 2005. URL: http://tools.ietf.org/html/rfc4086 (see pp. 7, 8, 41, 43).

# References

[15] *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology (NIST) Special Publication 800-90Ar1. 2015. URL: https://doi.org/10.6028/NIST.SP.800-90Ar1 (see pp. 10–13, 20, 22, 23, 30–33, 42).

[16] *Recommendation for the Entropy Sources Used for Random Bit Generation*. National Institute of Standards and Technology (NIST) Special Publication 800-90B. 2018. URL: https://doi.org/10.6028/NIST.SP.800-90B (see p. 9).