Security of Pseudo-Random Number Generators With Input

Damien Vergnaud

École normale supérieure – INRIA – PSL

wr0ng
April, 30th 2017

(with Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault & Daniel Wichs)

About this Talk

- examine randomness generation for cryptography
- give
  - *security definitions*
  - a *construction* meeting the formalized requirements.
- analyze
  - a previous construction proposed by Barak and Halevi in 2005
  - Linux random generators `/dev/random` and `/dev/urandom`

Contents

True Random Number Generators

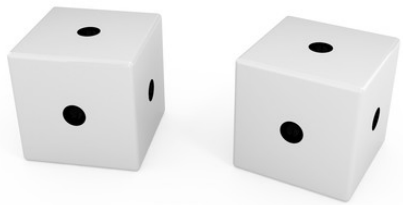
- Natural randomness in real world — previous talks

- Find a regular but random event and monitor
- but, need special hardware to do this
- but, often slow
- but, problems of bias or uneven distribution

True Random Number Generators

- Natural randomness in real world previous talks

- Find a regular but random event and monitor
- but, need special hardware to do this
- but, often slow
- but, problems of bias or uneven distribution



Random Sources and Extractors

- What kinds of random sources are useful ?
  - **impredictable** $\rightsquigarrow$ must have sufficient entropy
  - in cryptography: use min-entropy:

$$\mathbf{H}_\infty(X) = \min_{x \xleftarrow{\$} X} \{-\log \Pr[X = x]\}$$

- Build deterministic extractor ?
  - $f : \{0,1\}^n \to \{0,1\}$,
    s.t. for X over $\{0,1\}^n$ with $\mathbf{H}_\infty(X) \geq n - 1$, $\Pr[f(X) = 0] = 1/2$
  - cannot exist

- $\rightsquigarrow$ Randomness extractors
  - use a small family of functions
  - parametrized by a **seed**
  - in cryptography: **public or private** ?

Random Sources and Extractors

- What kinds of random sources are useful ?
  - **impredictable** $\rightsquigarrow$ must have sufficient entropy
  - in cryptography: use min-entropy:

$$\mathbf{H}_\infty(X) = \min_{x \xleftarrow{\$} X} \{-\log \Pr[X = x]\}$$

- Build deterministic extractor ?
  - $f : \{0,1\}^n \to \{0,1\}$,
    s.t. for X over $\{0,1\}^n$ with $\mathbf{H}_\infty(X) \geq n - 1$, $\Pr[f(X) = 0] = 1/2$
  - cannot exist

- $\rightsquigarrow$ Randomness extractors
  - use a small family of functions
  - parametrized by a **seed**
  - in cryptography: **public or private** ?

Random Sources and Extractors

- What kinds of random sources are useful ?
  - **impredictable** $\rightsquigarrow$ must have sufficient entropy
  - in cryptography: use min-entropy:

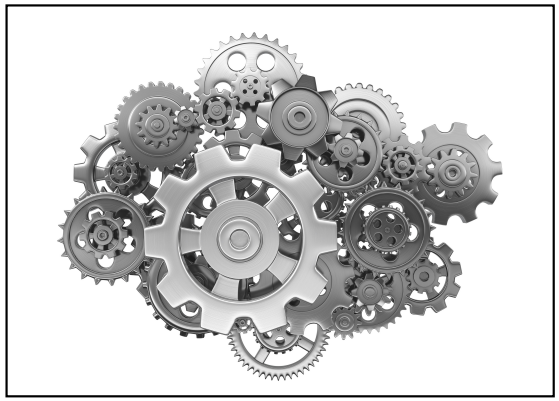
$$\mathbf{H}_\infty(X) = \min_{x \xleftarrow{\$} X} \{-\log \Pr[X = x]\}$$

- Build deterministic extractor ?
  - $f : \{0,1\}^n \to \{0,1\}$,
    s.t. for X over $\{0,1\}^n$ with $\mathbf{H}_\infty(X) \geq n - 1$, $\Pr[f(X) = 0] = 1/2$
  - cannot exist

- $\rightsquigarrow$ Randomness extractors
  - use a small family of functions
  - parametrized by a **seed**
  - in cryptography: **public or private** ?

(Deterministic) Pseudorandom Number Generators

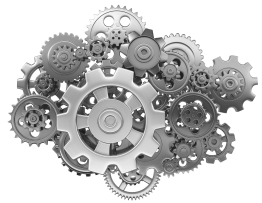
01101001001010010101101 10



01100010111101001010101111110101111010000101110...

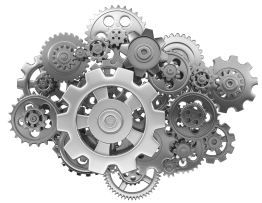
- output determined by a secret initial value
- output **approximates** the properties of random numbers
- fast and reproducible

Security of a PRNG



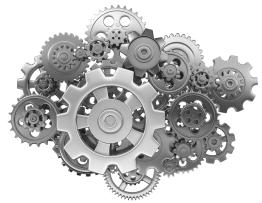
0110001011110100101010111111010111101000010111...

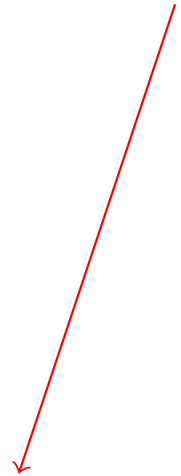
Security of a PRNG



0110001011110100101010111111010111101000010111...

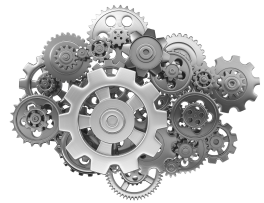
Security of a PRNG





01100010111101001010101111110101111010000010111...

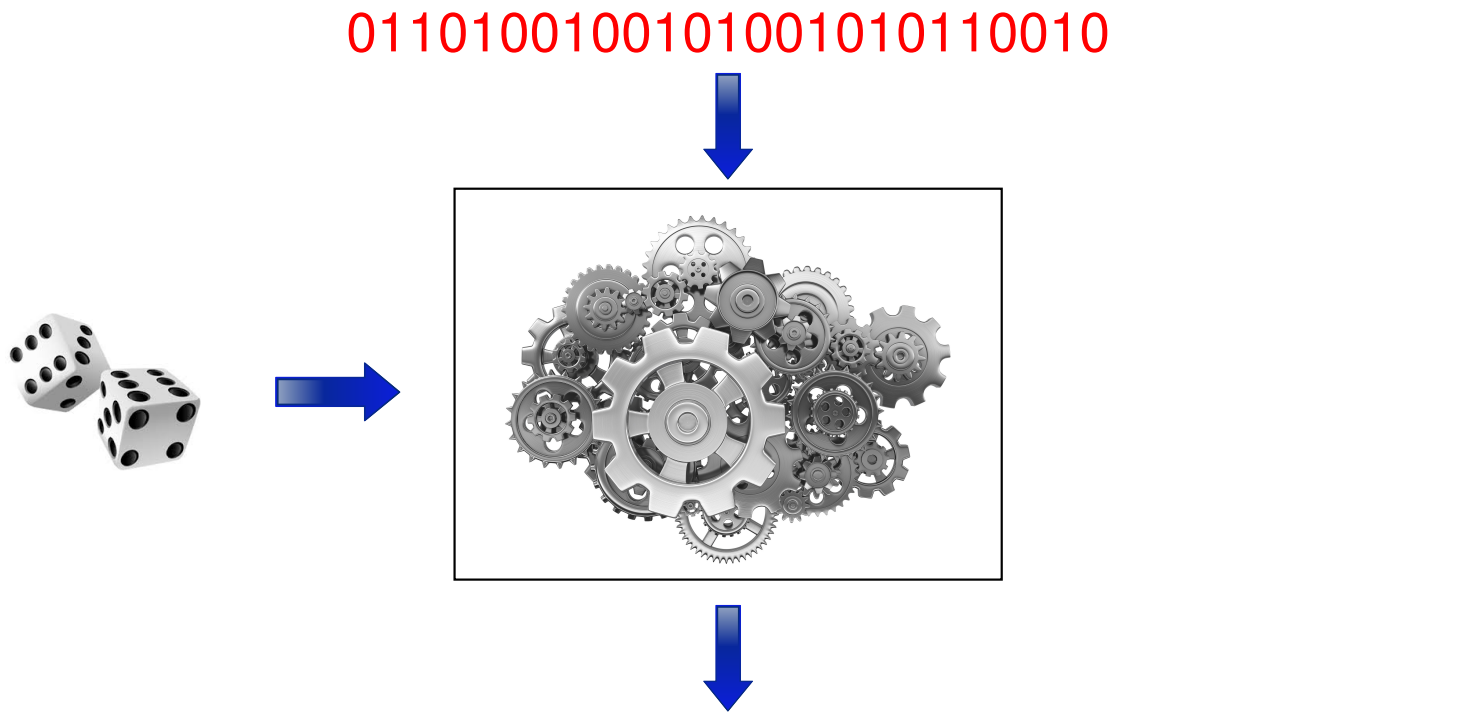
Security of a PRNG



01100010111101001010101111110101111010000010111...

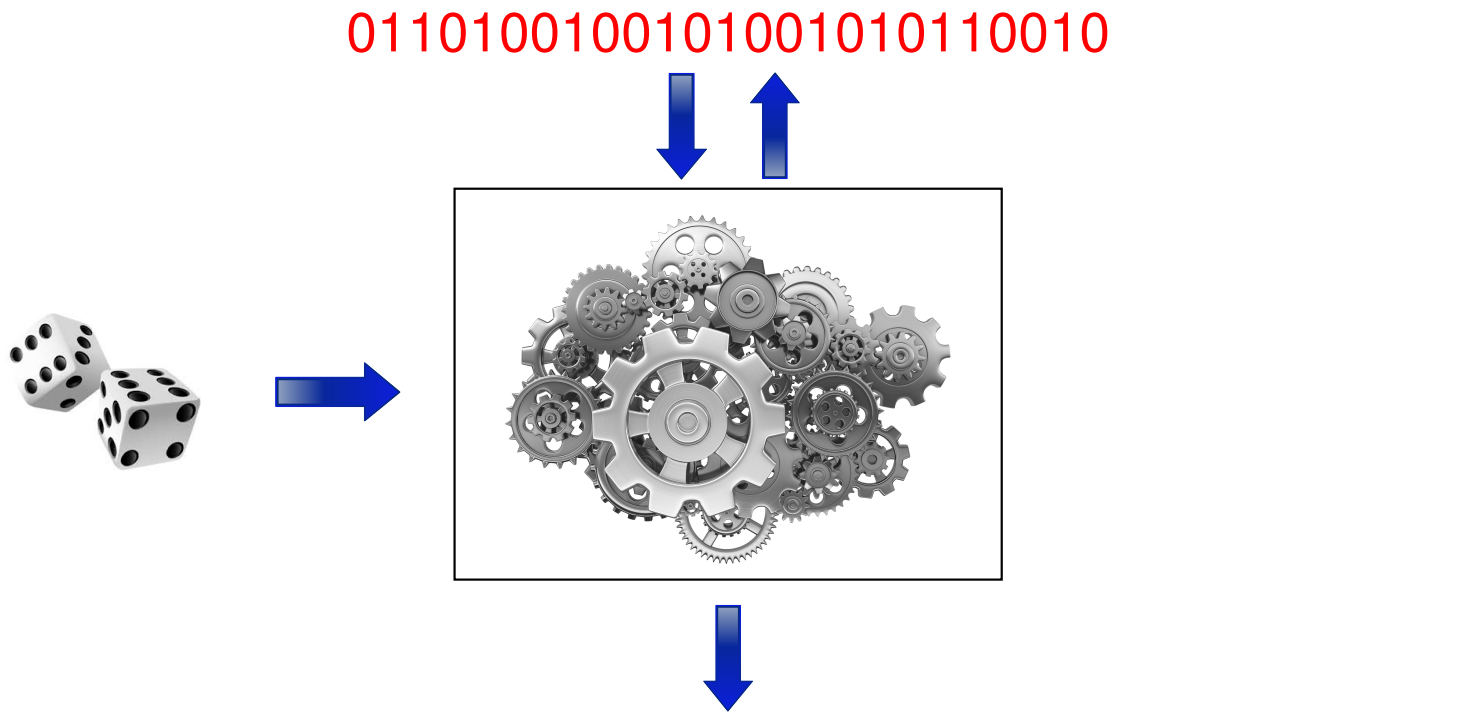
What if the key is compromised ?

Pseudorandom Number Generators with Inputs



- **Examples:**
  - Linux RNG : `/dev/random`, Yarrow, Fortuna, Havege, ...

Pseudorandom Number Generators with Inputs



- **Examples:**
  - Linux RNG : `/dev/random`, Yarrow, Fortuna, Havege, ...

Expected Security Properties

- Resilience: output looks random w/o knowledge of internal state
  - **Unknown/Known/Chosen input attacks**

- Security After State Compromise
  - **Forward security:**
    $\rightsquigarrow$ earlier output looks random with knowledge of current state
  - **Backward security:**
    $\rightsquigarrow$ future output looks random with knowledge of current state

How to formalize these security notions ?

Expected Security Properties

- Resilience: output looks random w/o knowledge of internal state
  - **Unknown/Known/Chosen input attacks**
- Security After State Compromise
  - **Forward security:**
    $\rightsquigarrow$ earlier output looks random with knowledge of current state
  - **Backward security:**
    $\rightsquigarrow$ future output looks random with knowledge of current state

How to formalize these security notions ?

Expected Security Properties

- Resilience: output looks random w/o knowledge of internal state
  - **Unknown/Known/Chosen input attacks**
- Security After State Compromise
  - **Forward security:**
    $\rightsquigarrow$ earlier output looks random with knowledge of current state
  - **Backward security:**
    $\rightsquigarrow$ future output looks random with knowledge of current state

How to formalize these security notions ?

Contents

Barak-Halevi Security Model (2005)

- $\mathcal{G} = (\textbf{refresh}, \textbf{next})$ is a PRNG with input
  - $\textbf{refresh}(S, I) = S' \in \{0,1\}^n$.
  - $\textbf{next}(S) = (S', R) \in \{0,1\}^n \times \{0,1\}^\ell$
- Security notion: Robustness

G_1

proc. good-refresh($\mathcal{D}$)
$x \xleftarrow{\$} \mathcal{D}$
$S \leftarrow \textbf{refresh}(S, x)$

proc. bad-refresh(x)
$S \leftarrow \textbf{refresh}(S, x)$

proc. set-state(S')
OUTPUT S
$S \leftarrow S'$

proc. next-ror()
$(R, S') \leftarrow \textbf{next}(S)$
$S \leftarrow S'$
OUTPUT R

G_2

proc. good-refresh($\mathcal{D}$)
$x \xleftarrow{\$} \mathcal{D}$
$S \leftarrow \textbf{refresh}(S, x)$
corrupt $\leftarrow$ false

proc. bad-refresh(x)
$S \leftarrow \textbf{refresh}(S, x)$

proc. set-state(S')
IF corrupt
OUTPUT S
ELSE
OUTPUT $\xleftarrow{\$} \{0,1\}^m$
$S \leftarrow S'$
corrupt $\leftarrow$ true

proc. next-ror()
$(R, S') \leftarrow \textbf{next}(S)$
$S \leftarrow S'$
IF corrupt
OUTPUT R
ELSE
OUTPUT $\xleftarrow{\$} \{0,1\}^\ell$

Barak-Halevi Security Model (2005)

- $\mathcal{G} = (\textbf{refresh}, \textbf{next})$ is a PRNG with input
  - $\textbf{refresh}(S, I) = S' \in \{0,1\}^n$.
  - $\textbf{next}(S) = (S', R) \in \{0,1\}^n \times \{0,1\}^\ell$
- Security notion: Robustness

G_1

proc. good-refresh($\mathcal{D}$)
$x \xleftarrow{\$} \mathcal{D}$
$S \leftarrow \textbf{refresh}(S, x)$

proc. bad-refresh(x)
$S \leftarrow \textbf{refresh}(S, x)$

proc. set-state(S')
OUTPUT S
$S \leftarrow S'$

proc. next-ror()
$(R, S') \leftarrow \textbf{next}(S)$
$S \leftarrow S'$
OUTPUT R

G_2

proc. good-refresh($\mathcal{D}$)
$x \xleftarrow{\$} \mathcal{D}$
$S \leftarrow \textbf{refresh}(S, x)$
corrupt $\leftarrow$ false

proc. bad-refresh(x)
$S \leftarrow \textbf{refresh}(S, x)$

proc. set-state(S')
IF corrupt
OUTPUT S
ELSE
OUTPUT $\xleftarrow{\$} \{0,1\}^m$
$S \leftarrow S'$
corrupt $\leftarrow$ true

proc. next-ror()
$(R, S') \leftarrow \textbf{next}(S)$
$S \leftarrow S'$
IF corrupt
OUTPUT R
ELSE
OUTPUT $\xleftarrow{\$} \{0,1\}^\ell$

Defects in Barak-Halevi Model

Entropy accumulation

- **null** or **high entropy** inputs,
- **but**, entropy could be accumulated slowly in S.
- a PRNG should recover from state compromise
  (if the amount of accumulated entropy crosses some threshold)

Need for a **setup** procedure

- deterministic randomness extractors do not exist!
- Two options:
  - restrict the family of permitted high-entropy distributions.
  - add a **setup** procedure which outputs some public parameters
    (used by **next** and **refresh**)

Defects in Barak-Halevi Model

Entropy accumulation

- **null** or **high entropy** inputs,
- **but**, entropy could be accumulated slowly in S.
- a PRNG should recover from state compromise
  (if the amount of accumulated entropy crosses some threshold)

Need for a **setup** procedure

- deterministic randomness extractors do not exist!
- Two options:
  - restrict the family of permitted high-entropy distributions.
  - add a **setup** procedure which outputs some public parameters
    (used by **next** and **refresh**)

Defects in Barak-Halevi Model

State Pseudorandomness

- BH model ensures that S is indistinguishable from random
- **But** technical parameters do not need to be random (e.g. Linux contains (predictable) entropy estimators).
- Pseudorandomness of the state is not actually a requirement
- **Only pseudorandomness of the output is !**

New Model Description

- $\mathcal{G} = (\textbf{setup}, \textbf{refresh}, \textbf{next})$ is a PRNG with input
  - **setup** output public parameters seed
  - **refresh**$(S, I) = S' \in \{0,1\}^n$.
  - **next**$(S) = (S', R) \in \{0,1\}^n \times \{0,1\}^\ell$

Adversary divided into two parts $(\mathcal{A}, \mathcal{D})$

- $\mathcal{D} : \sigma \to (\sigma', I, \gamma, z)$ is a legitimate distribution sampler
  - $\sigma =$ state of $\mathcal{D}$.
  - $I =$ next input for **refresh**
  - $\gamma =$ **entropy estimation** of I
  - $z =$ **leakage** about I given to $\mathcal{A}$
  - $\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_\mathcal{D}}, z_1, \ldots, z_{q_\mathcal{D}}, \gamma_1, \ldots, \gamma_{q_\mathcal{D}}) \geq \gamma_j$
- seed is not passed to $\mathcal{D}$ but is given to $\mathcal{A}$

Security Games

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,

corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,

$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,

$c \leftarrow 0$

OUTPUT R

proc. get-state

$c \leftarrow 0$, corrupt $\leftarrow$ true

OUTPUT S

proc. set-state(S^*)

$c \leftarrow 0$, corrupt $\leftarrow$ true

$S \leftarrow S^*$

Security Games

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,

$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,

corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,

$c \leftarrow 0$

OUTPUT R

proc. get-state

$c \leftarrow 0$, corrupt $\leftarrow$ true

OUTPUT S

proc. set-state(S^*)

$c \leftarrow 0$, corrupt $\leftarrow$ true

$S \leftarrow S^*$

Resilience

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,

corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,

$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,

$c \leftarrow 0$

OUTPUT R

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

Backward Security

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,
corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,
$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,
$c \leftarrow 0$

OUTPUT R

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

proc. set-state(S^*) (single first call)

$c \leftarrow 0$, corrupt $\leftarrow$ true

$S \leftarrow S^*$

Forward Security

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,

corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,

$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,

$c \leftarrow 0$

OUTPUT R

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

proc. get-state (single last call)

$c \leftarrow 0$, corrupt $\leftarrow$ true

OUTPUT S

Robustness

proc. initialize

seed $\xleftarrow{\$}$ **setup**; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$;

$c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$

OUTPUT seed

proc. finalize(b^*)

IF $b = b^*$ RETURN 1

ELSE RETURN 0

proc. next-ror

$(S, R_0) \leftarrow$ **next**(S)

$R_1 \xleftarrow{\$} \{0,1\}^\ell$

IF corrupt $=$ true,

$c \leftarrow 0$, RETURN R_0

ELSE RETURN R_b

proc. $\mathcal{D}$-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$

$S \leftarrow$ **refresh**(S, I)

$c \leftarrow c + \gamma$

IF $c \geq \gamma^*$,

corrupt $\leftarrow$ false

OUTPUT (γ, z)

proc. get-next

$(S, R) \leftarrow$ **next**(S)

IF corrupt $=$ true,

$c \leftarrow 0$

OUTPUT R

proc. get-state

$c \leftarrow 0$, corrupt $\leftarrow$ true

OUTPUT S

proc. set-state(S^*)

$c \leftarrow 0$, corrupt $\leftarrow$ true

$S \leftarrow S^*$

Barak-Halevi Construction

- Extract : $\{0,1\}^p \longrightarrow \{0,1\}^n$ a randomness extractor
- $\mathbf{G} : \{0,1\}^n \longrightarrow \{0,1\}^{n+\ell}$ a (deterministic) PRNG

Barak-Halevi Construction

- $\mathbf{refresh}(S, I) = [\mathbf{G}(S \oplus \mathrm{Extract}(I))]_1^n$
- $\mathbf{next}(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model

Simplified Barak-Halevi Construction

- $\mathbf{refresh}(S, I) = S \oplus \mathrm{Extract}(I)$
- $\mathbf{next}(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model (if one drops state pseudorandomness)

Barak-Halevi Construction

- Extract : $\{0,1\}^p \longrightarrow \{0,1\}^n$ a randomness extractor
- $\mathbf{G} : \{0,1\}^n \longrightarrow \{0,1\}^{n+\ell}$ a (deterministic) PRNG

Barak-Halevi Construction

- $\mathbf{refresh}(S, I) = [\mathbf{G}(S \oplus \mathrm{Extract}(I))]_1^n$
- $\mathbf{next}(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model

Simplified Barak-Halevi Construction

- $\mathbf{refresh}(S, I) = S \oplus \mathrm{Extract}(I)$
- $\mathbf{next}(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model (if one drops state pseudorandomness)

Barak-Halevi Construction

Simplified Barak-Halevi Construction

- **refresh**$(S, I) = S \oplus \text{Extract}(I)$
- **next**$(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model (if one drops state pseudorandomness)

- but, does not accumulate entropy!
- is not backward secure in [DPRVW13] model

- $\mathcal{D} : \sigma = \emptyset \rightarrow (\sigma', I, \gamma, z) = (\emptyset, b^p, 1, \emptyset)$ with $b \xleftarrow{\$} \{0,1\}$ is a (stateless) legitimate distribution sampler
- $\mathcal{A}$
  - calls set-state(0^n) $(S_0 = 0^n)$,
  - makes γ^* calls to $\mathcal{D}$-refresh $(S_j = \mathcal{D}\text{-refresh}(S_{j-1}, b^p))$
  - makes many calls to next-ror.

$Y(b) = \text{Extract}(b^p) \rightsquigarrow S_{2j} \in \{0^n, Y(0) \oplus Y(1)\}$ and $S_{2j+1} \in \{Y(0), Y(1)\}$

Barak-Halevi Construction

Simplified Barak-Halevi Construction

- **refresh**$(S, I) = S \oplus \text{Extract}(I)$
- **next**$(S) = \mathbf{G}(S)$

$\rightsquigarrow$ robust in BH model (if one drops state pseudorandomness)

- but, does not accumulate entropy!
- is not backward secure in [DPRVW13] model

- $\mathcal{D} : \sigma = \emptyset \to (\sigma', I, \gamma, z) = (\emptyset, b^p, 1, \emptyset)$ with $b \xleftarrow{\$} \{0,1\}$ is a (stateless) legitimate distribution sampler
- $\mathcal{A}$
  - calls set-state(0^n) $(S_0 = 0^n)$,
  - makes γ^* calls to $\mathcal{D}$-refresh $(S_j = \mathcal{D}\text{-refresh}(S_{j-1}, b^p))$
  - makes many calls to next-ror.

$Y(b) = \text{Extract}(b^p) \rightsquigarrow S_{2j} \in \{0^n, Y(0) \oplus Y(1)\}$ and $S_{2j+1} \in \{Y(0), Y(1)\}$

Contents

A Provably Secure Construction

- $\mathbf{G} : \{0,1\}^m \longrightarrow \{0,1\}^{n+\ell}$ a (deterministic) PRNG

Construction

- **setup**$(\cdot) =$ seed $= (X, X') \xleftarrow{\$} \{0,1\}^{2n}$.
- **refresh**$(S, I) = S \cdot X + I \in \mathbb{F}_{2^n}$.
- **next**$(S) = \mathbf{G}([X' \cdot S]_1^m)$.

- it **preserves security**
- it **accumulates entropy**
- $\rightsquigarrow$ robust in [DPRVW13] model !

A Provably Secure Construction

Lemma 1

This construction **preserves security**.

- if the state S_0 starts uniformly random and uncompromised,
- and is refreshed with (adversarial) samples $I_1, \ldots, I_d \rightsquigarrow S_d$,
- $(S', R) = \textbf{next}(S_d)$

then R looks indistinguishable from uniform

Proof.

$$S_d := S \cdot X^d + I_{d-1} \cdot X^{d-1} + \cdots + I_1 \cdot X + I_0.$$

A Provably Secure Construction

Lemma 1

This construction **preserves security**.

- if the state S_0 starts uniformly random and uncompromised,
- and is refreshed with (adversarial) samples $I_1, \ldots, I_d \rightsquigarrow S_d$,
- $(S', R) = \textbf{next}(S_d)$

then R looks indistinguishable from uniform

Proof.

$$S_d := S \cdot X^d + I_{d-1} \cdot X^{d-1} + \cdots + I_1 \cdot X + I_0.$$

A Provably Secure Construction

Lemma 2

This construction **accumulates entropy**.

- if the state S_0 starts is compromised to some arbitrary value
- and is refreshed with $\mathcal{D}$-refresh samples $I_1, \ldots, I_d \rightsquigarrow S_d$,
- $(S', R) = \textbf{next}(S_d)$

then R looks indistinguishable from uniform

Proof.

$$h^*_{X,X'}(\bar{I}) := \left[X' \cdot \sum_{j=0}^{d-1} I_j \cdot X^j \right]_1^m .$$

is $2^{-m}(1 + d \cdot 2^{m-n})$-universal. □

A Provably Secure Construction

Lemma 2

This construction **accumulates entropy**.

- if the state S_0 starts is compromised to some arbitrary value
- and is refreshed with $\mathcal{D}$-refresh samples $I_1, \ldots, I_d \rightsquigarrow S_d$,
- $(S', R) = \textbf{next}(S_d)$

then R looks indistinguishable from uniform

Proof.

$$h^*_{X,X'}(\bar{I}) := \left[X' \cdot \sum_{j=0}^{d-1} I_j \cdot X^j \right]_1^m .$$

is $2^{-m}(1 + d \cdot 2^{m-n})$-universal. □

Contents

The Linux Random Number Generator

- part of the **Linux kernel** since 1994
- from Theodore Ts'o and Matt Mackall

- only definition in the code (with comments) :
  - About 1700 lines

- **Previous Analysis:**
  - Barak-Halevi, 2005: almost no mentioning of the Linux RNG
  - Gutterman-Pinkas-Reinman, 2006: some weaknesses
  - Lacharme-Röck-Strubel-Videau, 2012: detailed description

- Two different versions :
  - `/dev/random`: limits the number of bits by the estimated entropy
  - `/dev/urandom`: generates as many bits as the user asks for

The Linux Random Number Generator

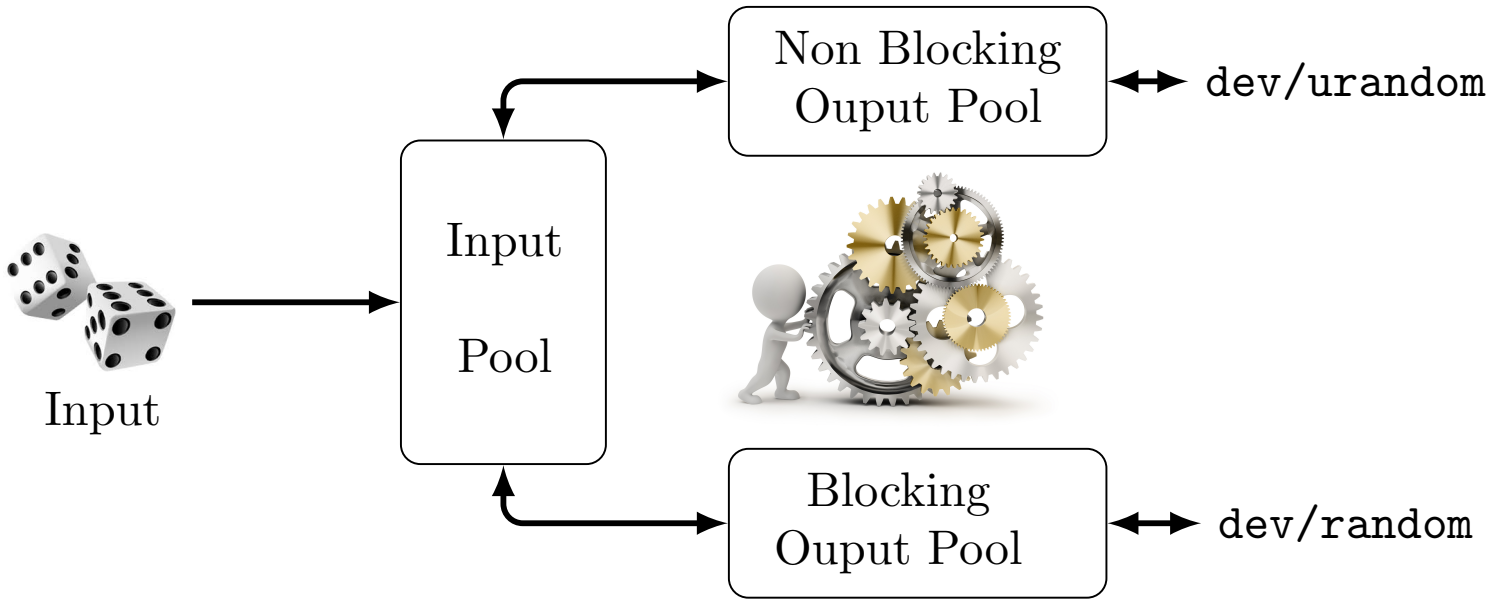
- part of the **Linux kernel** since 1994
- from Theodore Ts'o and Matt Mackall

- only definition in the code (with comments) :
  - About 1700 lines

- **Previous Analysis:**
  - Barak-Halevi, 2005: almost no mentioning of the Linux RNG
  - Gutterman-Pinkas-Reinman, 2006: some weaknesses
  - Lacharme-Röck-Strubel-Videau, 2012: detailed description

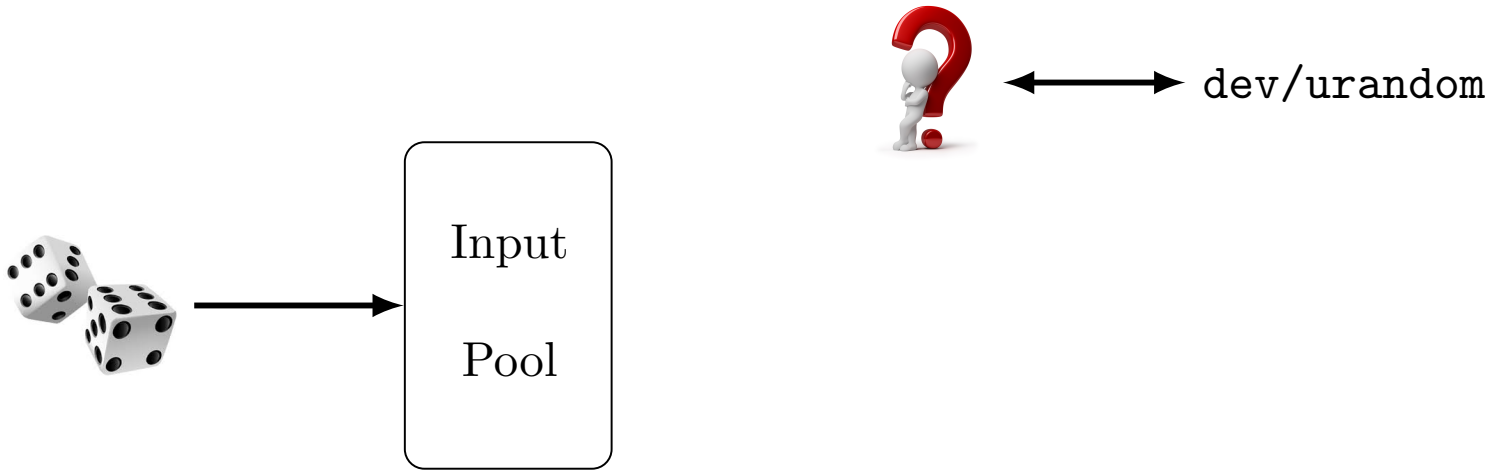
- Two different versions :
  - `/dev/random`: limits the number of bits by the estimated entropy
  - `/dev/urandom`: generates as many bits as the user asks for

General Overview of LINUX PRNG



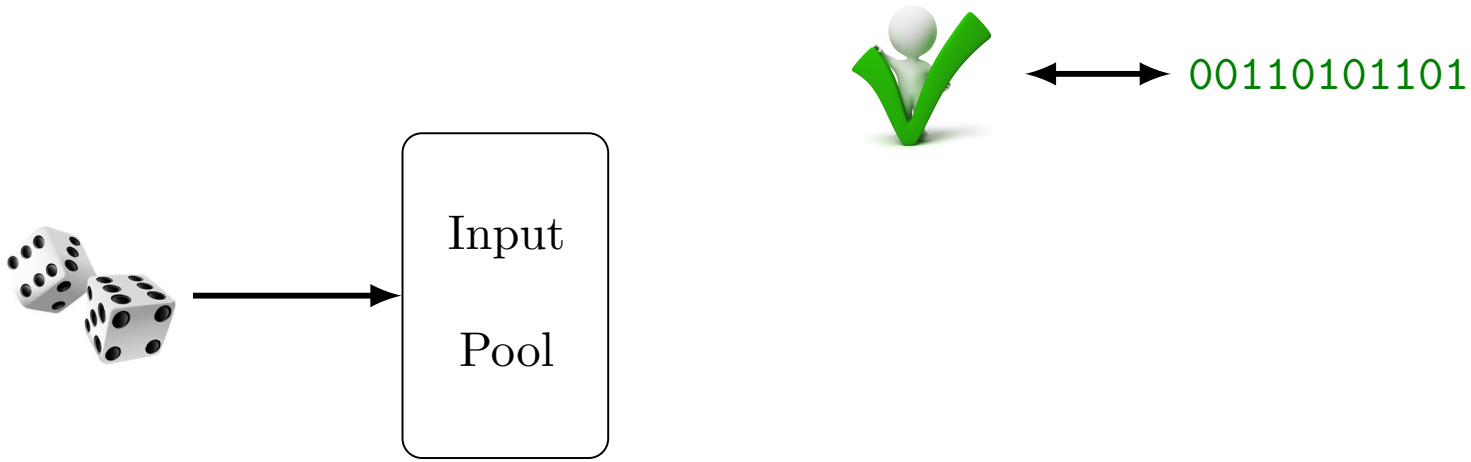
- $|I| = 96$, $|S| = 6144$, $|R| = 80$
- **refresh** and **next** uses a Mixing function and a Hash function
- all transfers between pools rely on Entropy Estimators

dev/urandom Output Request



- Is there enough entropy in Non Blocking Output Pool ?

/dev/urandom Output Request

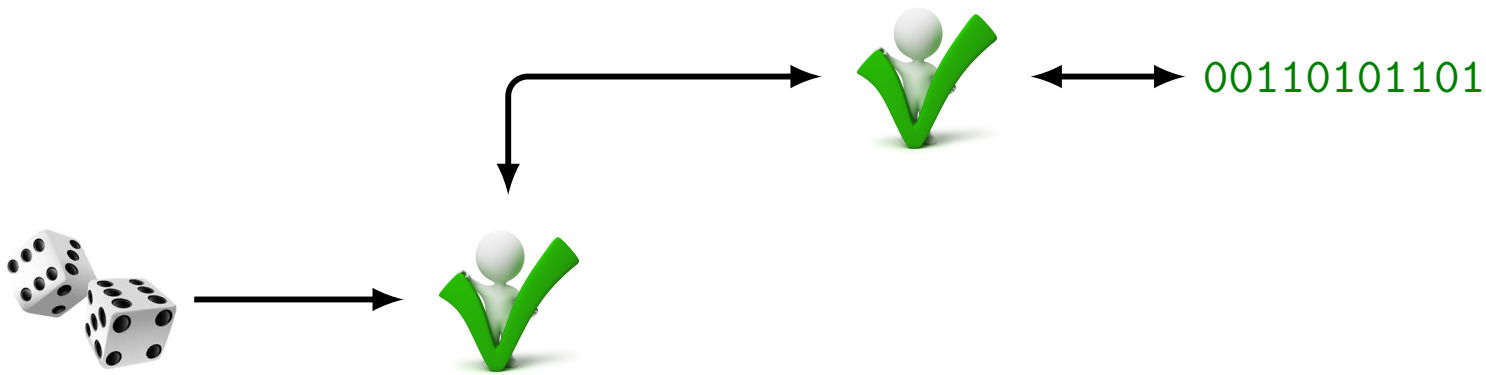


- Is there enough entropy in output pool ?
- Yes, output the requested bytes !

dev/urandom Output Request

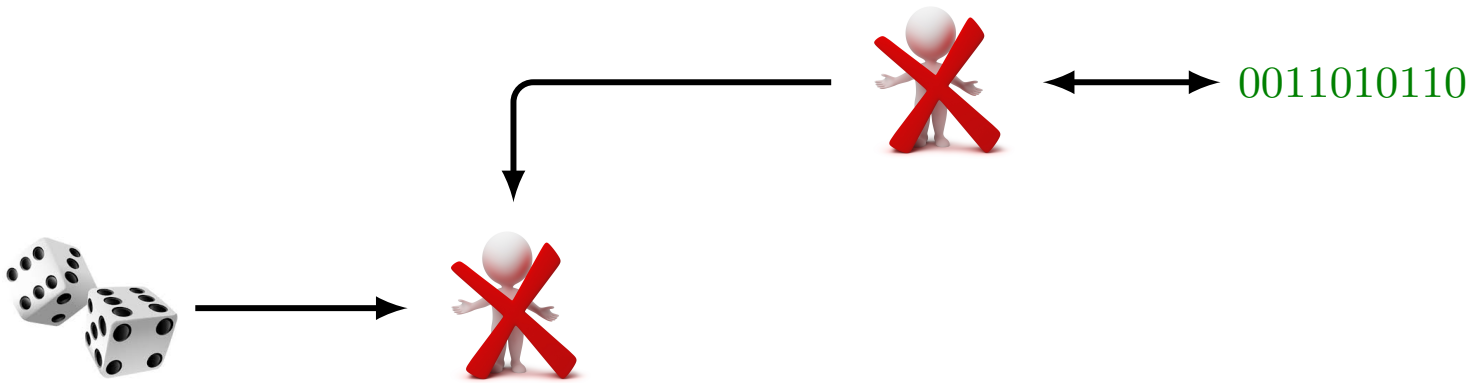
- Is there enough entropy in output pool ?
- No, ask the input pool !
  - Is there enough entropy in input pool ?

/dev/urandom Output Request



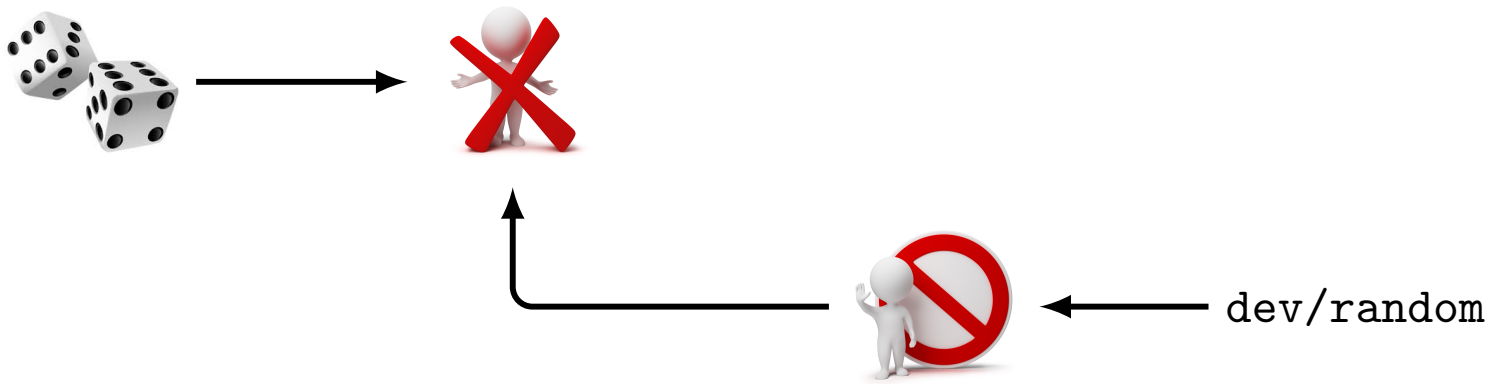
- Is there enough entropy in output pool ?
- No, ask the input pool !
  - Is there enough entropy in input pool ?
  - Yes, transfer from input pool to output pool and generate!

/dev/urandom Output Request



- Is there enough entropy in output pool ?
- No, ask the input pool !
  - Is there enough entropy in input pool ?
  - No, generate output anyway !

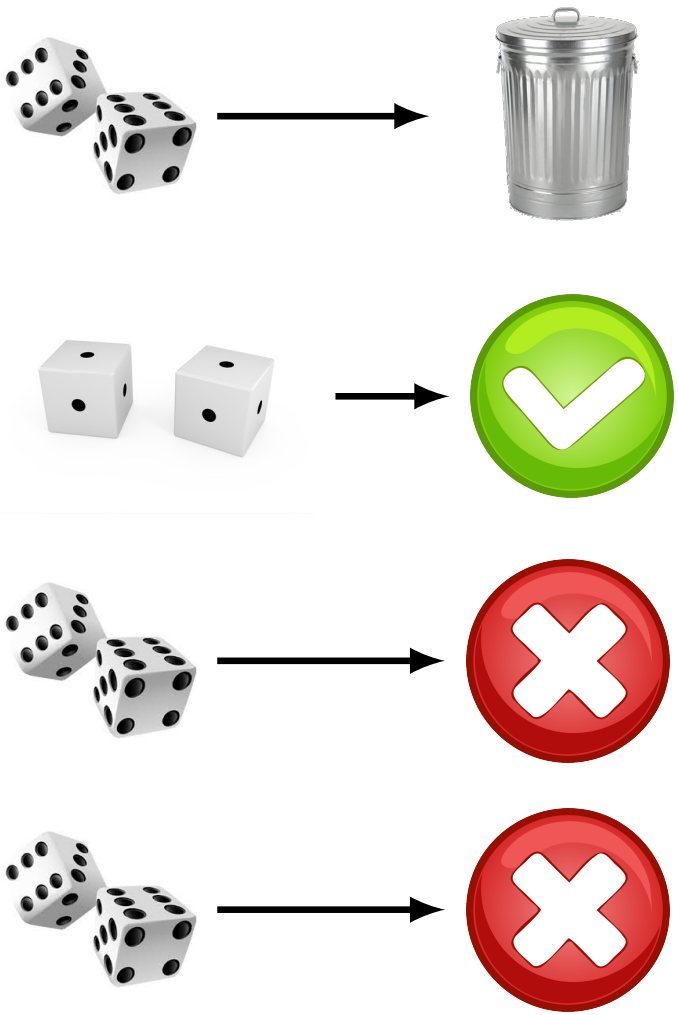
Difference with `dev/random`



- Is there enough entropy in output pool ?
- No, ask the input pool !
  - Is there enough entropy in input pool ?
  - No, do not generate output and wait !

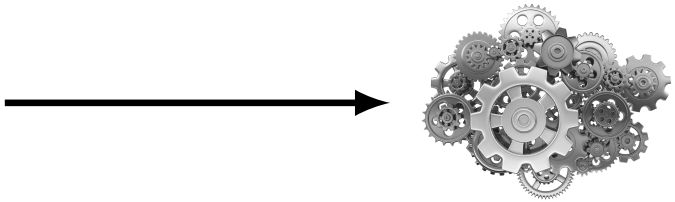
Defects of LINUX PRNG

- if input pool contains enough entropy, don't **refresh** (before [DPRVW13])
- there exists $\mathcal{D}_0$, $\mathbf{H}_\infty(\mathcal{D}_0) = 0$, that LINUX estimates high
- there exists $\mathcal{D}_1$, $\mathbf{H}_\infty(\mathcal{D}_1) = 64$, that LINUX estimates 0
- there exists $\mathcal{D}_2$, $\mathbf{H}_\infty(\mathcal{D}_2) = 1$, for which LINUX does not accumulate

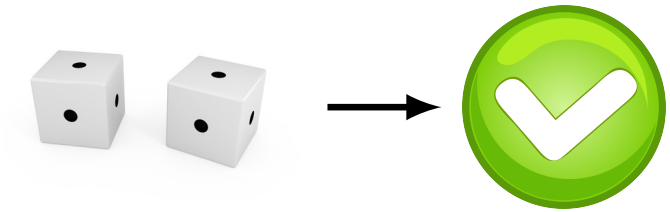


`dev/random` was not Robust

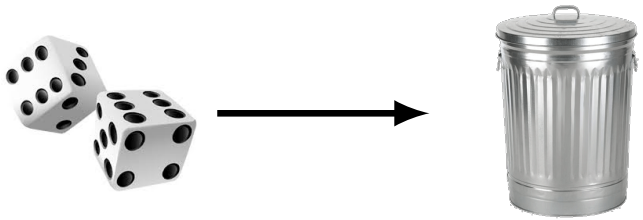
- first step : get-state



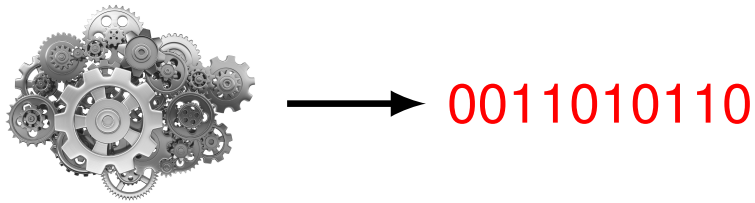
- $\mathcal{D}$-refresh with $\mathcal{D}_0$ ($\mathbf{H}_\infty = 0$), until input pool is full



- $\mathcal{D}$-refresh with $\mathcal{D}_1$ ($\mathbf{H}_\infty = 64$), which are ignored

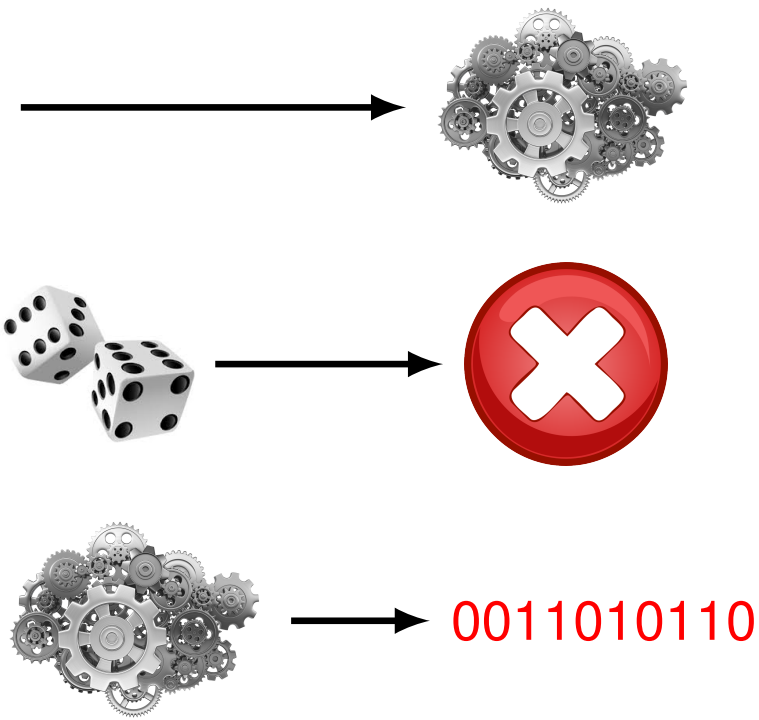


- **next**: $\mathbf{H}_\infty(R) = 0$



`dev/urandom` was not Robust

- first step : get-state
- $\mathcal{D}$-refresh with $\mathcal{D}_1$ ($\mathbf{H}_\infty = 64$), which are not transfered
- **next** : $\mathbf{H}_\infty(R) = 0$



Contents

Follow-up Works

- **Other Attacks**
  (Cornejo-Ruhault – ACM CCS 2014)
- **Security against Premature Next**
  (Dodis, Shamir, Stephens-Davidowitz, Wichs – Crypto 2014)
- **Analysis of Intel's Secure Key RNG**
  (Shrimpton, Terashima – Eurocrypt 2015)
- **Backdoored PRNGs**
  (Degabriele, Paterson, Schuldt, Woodage – Crypto 2016)
  Kenny's talk ...
- **Sponge-Based PRNGs**
  (Gaži, Tessaro – Eurocrypt 2016)
  see next talk ...

Conclusion

Generation of random numbers is too important to be left to chance . . .

- Analysis of BH model and construction.
- DPRVW13 security model for PRNG with input.
- Attacks on LINUX PRNGs
  - using entropy estimator
  - using mixing function (see paper)
- Construction provably secure and efficient.