

# Cryptographic Engineering: Random Number Generation

October 2021  
Sujoy Sinha Roy  
[sujoy.sinharoy@iaik.tugraz.at](mailto:sujoy.sinharoy@iaik.tugraz.at)



1. Choose two large primes randomly:  $p$  and  $q$
2. Modulus:  $m = p * q$
3. Public key:  
 $e = \text{random co-prime to } \phi(m) = (p-1) * (q-1)$
4. Private key:  
 $d = e^{-1} \text{ mod } \phi(m)$

# Crypto protocols need randomness

Example: The RSA Key Generation Algorithm:

1. Choose two large primes **randomly**:  $p$  and  $q$
2. Modulus:  $m = p * q$
3. Public key:  
 $e = \text{random}$  co-prime to  $\phi(m) = (p-1) * (q-1)$
4. Private key:  
 $d = e^{-1} \text{ mod } \phi(m)$

**How do we generate high quality randomness for crypto?**

## *rand()* function in C

```
int main(){  
  
    ....  
    int rand_buffer[N]  
  
    for (int j = 0; j < N; j++)  
        rand_buffer[j] = rand();  
  
    do_cryptography(rand_buffer);  
    ...  
}
```

Is this secure?

## *rand()* and *srand()* functions in C

```
int main(){  
  
    ....  
    int rand_buffer[N]  
  
    srand(time(NULL)); // randomize seed  
  
    for (int j = 0; j < N; j++)  
        rand_buffer[j] = rand();  
  
    do_cryptography(rand_buffer);  
    ...  
}
```

Is this secure?

## Problems with C *rand()*

1. Produces pseudorandom sequences of not long-enough cycle.

```
int rand(void);
```

 → Can produce a maximum of  $2^{32}$  random integers only

### DESCRIPTION

```
The rand() function returns a pseudo-random integer in the range 0 to RAND_MAX inclusive (i.e., the mathematical range [0, RAND_MAX]).
```

Source: man page of *srand*

2. The number of possible seeds is  $2^{32}$  only.

```
void srand(unsigned int seed);
```

 Type is 'int'.

3. Legacy *rand()* function implementations produced much less randomness for the low-order bits.

# Attacks exploiting weak randomness

## Cryptanalysis of the Random Number Generator of the Windows Operating System

Leo Dorrendorf

School of Engineering and Computer Science  
The Hebrew University of Jerusalem  
91904 Jerusalem, Israel  
dorrel@cs.huji.ac.il

Zvi Gutterman

School of Engineering and Computer Science  
The Hebrew University of Jerusalem  
91904 Jerusalem, Israel  
zvikag@cs.huji.ac.il

Benny Pinkas\*

Department of Computer Science  
University of Haifa  
31905 Haifa, Israel  
benny@pinkas.net

November 4, 2007

### Abstract

The pseudo-random number generator (PRNG) used by the Windows operating system is the most commonly used PRNG. The pseudo-randomness of the output of this generator is crucial for the security of almost any application running in Windows. Nevertheless, its exact algorithm was never published.

# Attacks exploiting weak randomness

The New York Times

## *Flaw Found in an Online Encryption Method*



By [John Markoff](#)

Feb. 14, 2012

SAN FRANCISCO — A team of European and American mathematicians and cryptographers have discovered an unexpected weakness in the encryption system widely used worldwide for online shopping, banking, e-mail and other Internet services intended to remain private and secure.

The flaw — which involves a small but measurable number of cases — has to do with the way the system generates random numbers, which are used to make it practically impossible for an attacker to unscramble digital messages. While it can affect the

Lenstra, Hughes, Augier, Bos, Kleinjung, and Wachter were able to factor 0.2% of deployed RSA keys using simple Euclid's algorithm.



# How to generate random numbers for cryptography?

In Unix-based OS, you get high-entropy randomness by reading `/dev/random` or `/dev/urandom` files.

→ These files collect noise from the environment (e.g., device driver noise, mouse movements, key-board timings, etc.)

E.g., reading 10 random bytes from `/dev/random`

```
% hexdump -C -n 10 /dev/random
00000000  8c 31 12 e7 a5 31 e6 a3  af f9
0000000a
```

```
void randombytes(uint8_t *out, size_t outlen) {
    static int fd = -1;
    ssize_t ret;

    while(fd == -1) {
        fd = open("/dev/urandom", O_RDONLY);
        if(fd == -1 && errno == EINTR)
            continue;
        else if(fd == -1)
            abort();
    }

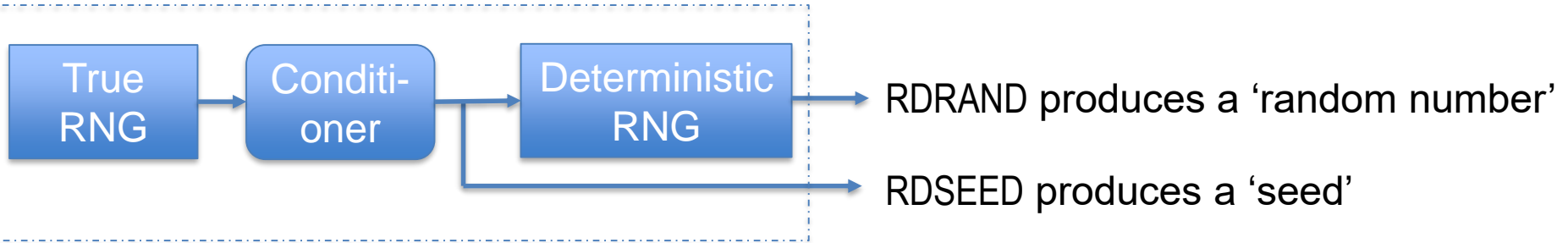
    while(outlen > 0) {
        ret = read(fd, out, outlen);
        if(ret == -1 && errno == EINTR)
            continue;
        else if(ret == -1)
            abort();

        out += ret;
        outlen -= ret;
    }
}
```

Example: Reading random bytes from /dev/urandom inside a C function.

# RDRAND and RDSEED instructions on new Intel processors

- These two instructions return random numbers from an on-chip Random Number Generator (RNG).



- RDSEED is used to produce a high-entropy 'seed'. This seed can be used to initialize any Pseudo Random Number Generator algo.
- RDRAND is used for generating many random numbers by using a deterministic-RNG with a random seed. The seed is changed periodically.

## RDRAND and RDSEED intrinsics for gcc

```
int _rdrand16_step(uint16_t*);  
int _rdrand32_step(uint32_t*);  
int _rdrand64_step(uint64_t*);
```

```
int _rdseed16_step(uint16_t*);  
int _rdseed32_step(uint32_t*);  
int _rdseed64_step(uint64_t*);
```

Pointer to the `uintXX_t` where the number/seed will be stored.

The functions return 1 when they succeed in generating a random number/seed. Otherwise they return a different value.

## Example C code: RDRAND and RDSEED

```
#include <stdio.h>
#include <immintrin.h>

int main() {
    unsigned long long result = 0ULL;

    int rc = _rdrand64_step (&result);

    printf("%i %llu\n", rc, result);

    return (rc != 1);
}
```

Compilation command:  
gcc -m64 -mrdrand <filename>.c

```
#include <stdio.h>
#include <immintrin.h>

int main() {
    unsigned long long result = 0ULL;

    int rc = _rdseed64_step (&result);

    printf("%i %llu\n", rc, result);

    return (rc != 1);
}
```

Compilation command:  
gcc -m64 -mrdseed <filename>.c

# Security 'debates' regarding RDRAND/RDSEED

The New York Times

## *N.S.A. Able to Foil Basic Safeguards of Privacy on Web*



By Nicole Perlroth, Jeff Larson and Scott Shane

Sept. 5, 2013

The National Security Agency is winning its long-running secret war on encryption, using supercomputers, technical trickery, court orders and behind-the-scenes persuasion to undermine the major tools protecting the privacy of everyday communications in the Internet age, according to newly disclosed documents.

The agency has circumvented or cracked much of the encryption, or digital scrambling, that guards global commerce and banking systems, protects sensitive data like trade secrets and medical records, and automatically secures the e-mails, Web searches,



Theodore Ts'o ▸ Public

Sep 5, 2013



I am so glad I resisted pressure from Intel engineers to let /dev/random rely only on the RDRAND instruction. To quote from the article below:

"By this year, the Sigint Enabling Project had found ways inside some of the encryption chips that scramble information for businesses and governments, either by working with chipmakers to insert back doors...."

Relying solely on the hardware random number generator which is using an implementation sealed inside a chip which is impossible to audit is a **BAD** idea.

sed to:

insert vulnerabilities  
on devices used

collect target network  
networks.

N.S.A. Foils Much Internet Encryption  
nytimes.com



246



626



272

<https://web.archive.org/web/20180611180213/https://plus.google.com/117091380454742934025/posts/SDcoemc9V3J>

**Next part: How to design your own True-RNG in HW?**

# Classification of Random Number Generators

Random number generators (RNG) can be classified into two main types.

1. True Random Number Generator (TRNG)
  - Also known as non-deterministic RNG
  - Produces true random numbers
  - Source of randomness: unpredictable processes
2. Pseudo Random Number Generator (PRNG)
  - Also known as deterministic RNG
  - Expands a short seed into a long string using a deterministic algo.
  - Does not produce any *\*new\** randomness

**We will mainly discuss implementations of TRNGs**



# **Design and Analysis of TRNGs**

# High-level diagram of TRNG



- 1. Entropy source** is the component where unpredictable physical processes run.
  - There are different types of physical processes that can be used
  - Combination of them is also possible
  - Produces time-continuous analog output
- 2. Digitization** is the component that samples analog output of the entropy source.
  - Produces binary bits
  - Sampling frequency influences the quality of randomness

# Different Entropy Sources for TRNGs

1. Thermal noise

**2. Timing jitter**  We will study this type of TRNGs.

3. Quantum effect

4. Metastability

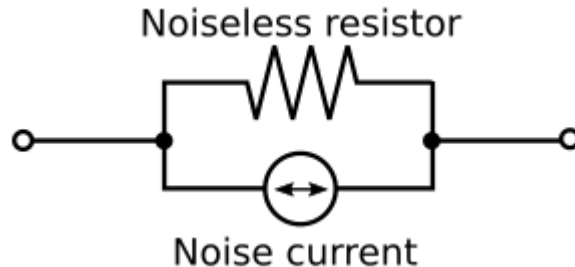
5. ... any combination of them

# Entropy Sources: Thermal Noise

- Generated by the thermal agitation of the charge carriers inside an electrical conductor → Present in all electronic devices
- Noise source is modelled as a 'current source' in parallel to a resistor

$$i_n = \sqrt{\frac{4k_B T \Delta f}{R}}$$

where  $k_B$  is Boltzmann's constant,  $T$  is absolute temperature,  $R$  is resistor value, and  $\Delta f$  is bandwidth over which noise is measured.



# Entropy Sources: Thermal Noise TRNG

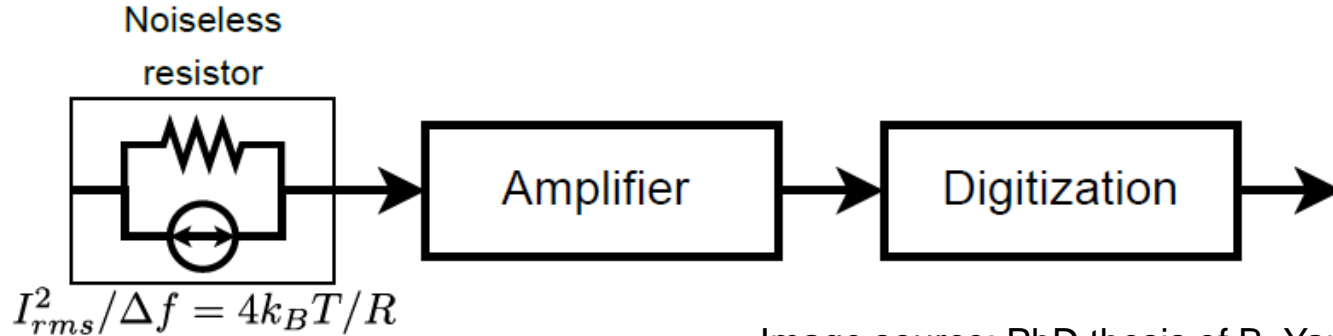
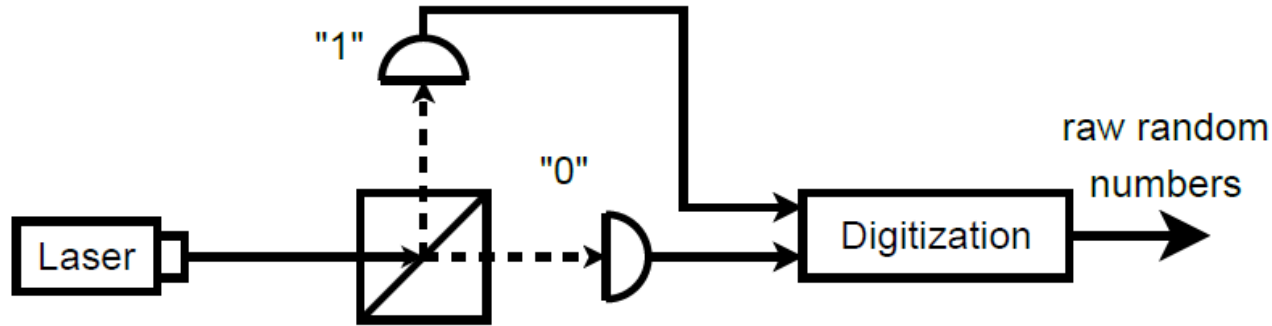


Image source: PhD thesis of B. Yang.

- Thermal noise over a resistor is first amplified
- Amplified noise is used to drive a Voltage Controlled Oscillator (VCO)
- Output of VCO is sampled (i.e., digitized) to produce random bits
- This kind of TRNGs are suitable for ASIC platforms
- Example: First proposed by Intel in [JK99]

# Entropy Sources: Quantum TRNG



Photons pass through a balanced beam splitter with equal transmissivity and reflectivity, and reach one of these two detectors. The results are encoded to the raw random numbers using digitization.

Photon source: Laser or Light emitting diode

There are commercially available ASIC chips of this type of TRNG.

# Entropy Sources: Timing Jitter

... we study this type of entropy source in detail

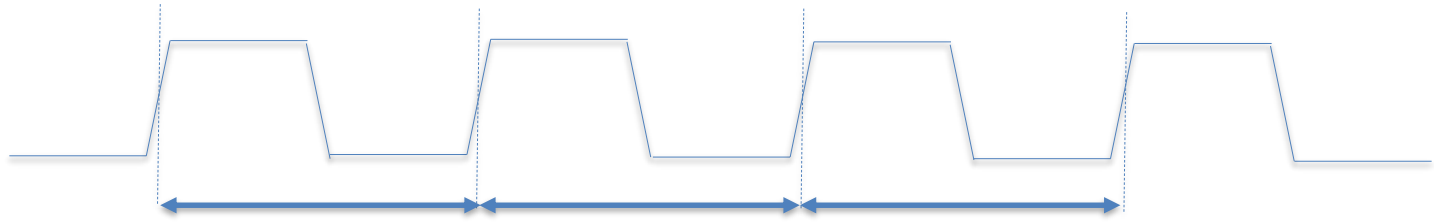


What do we call this waveform?

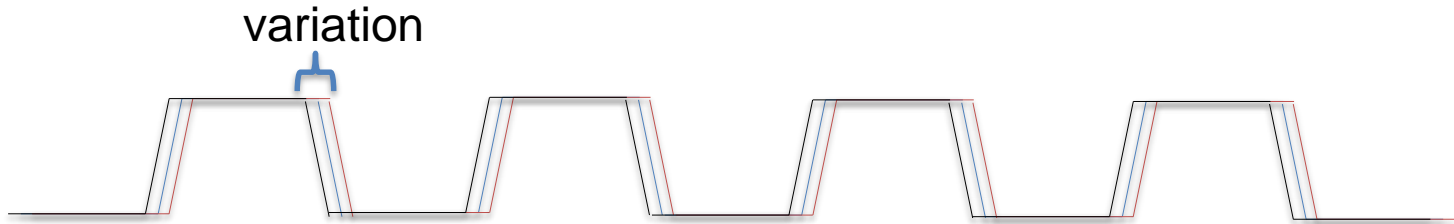
We call it the 'clock' in digital circuits

Actually it is an *ideal* clock





Ideal clock: All cycles are of fixed and equal length

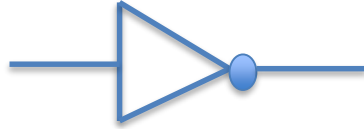


Real clock: Cycle lengths change

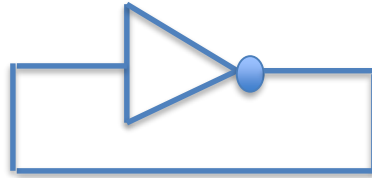
**Timing Jitter** is the deviation from true periodicity of a periodic signal.

## **How to use (random) jitter to produce true random numbers?**

1. The first step will be to create a periodic clock signal.
2. Next, sample from the 'jitter' region of the periodic signal.
3. Finally, digitize sampled values to produce random bits.



This is an inverter, e.g., a NOT gate

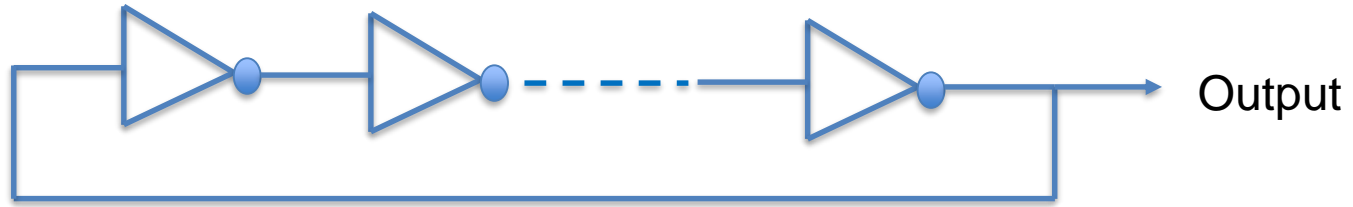


What happens with this configuration?

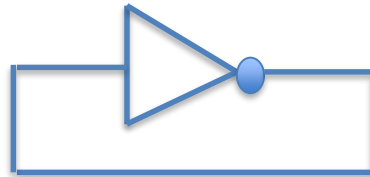
Answer: The output oscillates.

$\rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \dots$  and so on in a periodic manner

# Ring Oscillator (RO)



Any odd  $n$  number of inverters chained in a ring (i.e., a loop)



Special case with  $n = 1$

As  $n$  is odd, the output oscillates

...  $\rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$  ... and so on in a periodic manner



The average period is determined by the delay and number of inverter(s).

# Why do we see Jitter in a Ring Oscillator?

Delay of each logic element has two components:

1. a fixed component
2. and a variable component

The variable component is due to various noise sources in the device


- Global noise from the power supply
- Environmental noise (e.g., temperature, humidity, etc.)
- Correlated noise (e.g., flicker noise, telegraph noise)
- White noise, also known as Gaussian noise

# Why do we see Jitter in a Ring Oscillator?

Delay of each logic element has two components:

1. a fixed component
2. and a variable component

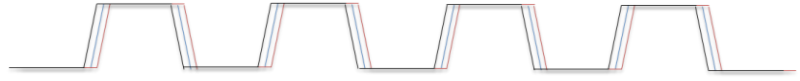
The variable component is due to various noise sources in the device

- Global noise from the power supply
  - Environmental noise (e.g., temperature, humidity, etc.)
  - Correlated noise (e.g., flicker noise, telegraph noise)
  - **White noise, also known as Gaussian noise** → Used for TRNG
- 
- Deterministic

# Jitter in Ring Oscillator

The (variable) period of a ring oscillator is given by

$$T_{RO} = T_0 + T_G + T_E + T_{corr} + T_{Gauss}$$



Only this component is presumed to be non-deterministic

Where

$T_0$  : average period of the RO

$T_G$  : contribution from global noise

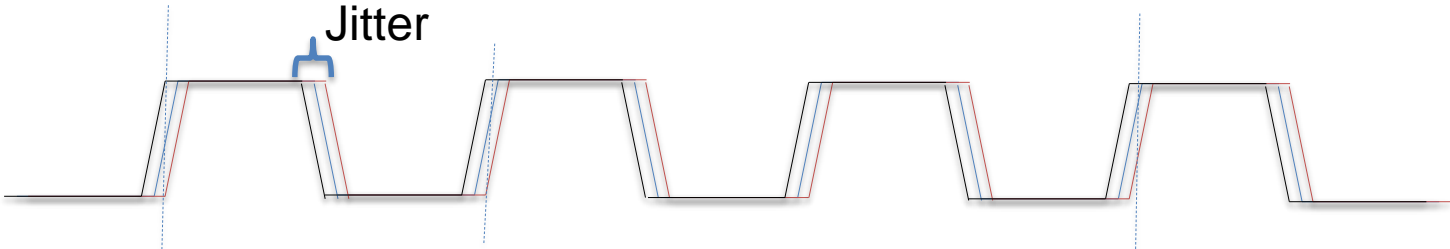
$T_E$  : Contribution from environmental noise

$T_{corr}$  : Contribution from correlated noise

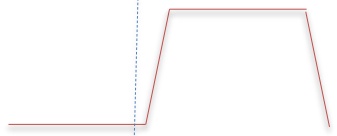
$T_{Gauss}$  : Contribution from Gaussian noise

# From jitter to random bits

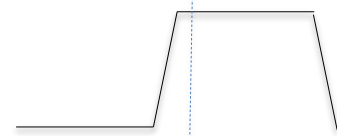
The transition region of RO output is unpredictable due to jitter.



- Sample the output of RO in this unpredictable region
- Digitize the sampled value to get a random bit.



Due to jitter, transition from 0→1 happens **after** the sampling point. Hence, 0 is sampled.

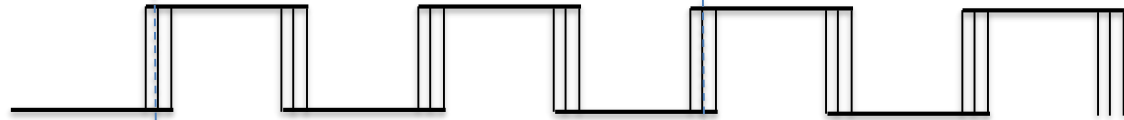


Due to jitter, transition from 0→1 happens **before** the sampling point. Hence, 1 is sampled.

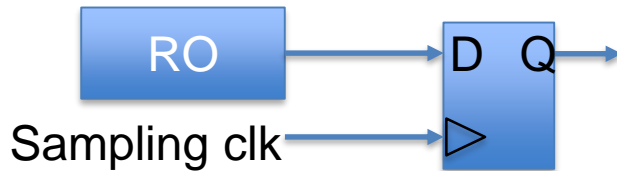


# Sampling jitter

Output of RO



Sampling clock



```
always @(posedge sampling_clk)
    D_ff <= RO_out_bit;
```

Verilog snippet for sampling

Such an implementation will work only if **posedge** transitions of the sampling clock coincide with the jitter regions of the RO output.

- ➔ Sampling clock and RO output must be in the same phase.
- ➔ Sampling clock can be output of another RO

# Practical problems with sampling jitter

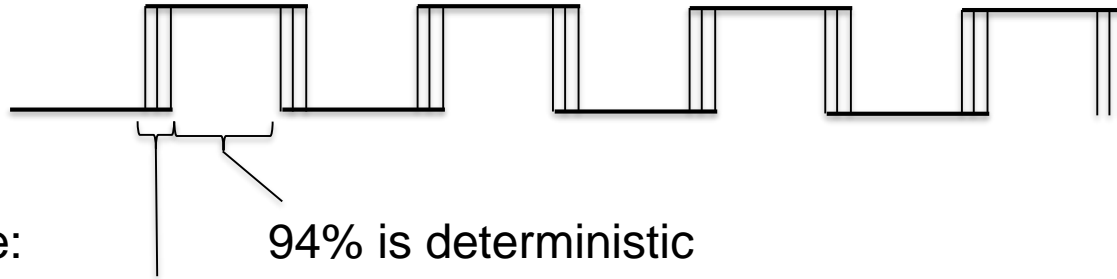
The previous configuration for sampling jitter does not work in practice.

- Exactly synchronizing two oscillating signals is very difficult to implement on digital platforms as that requires a special layout
- Moreover, with time two signals may drift from each other due to their own jitter and noises in the system.



# Solving the practical problem of sampling jitter

**Cause of failure:** The previous approach fails because width of the **nondeterministic** region is really small w.r.t deterministic region  
→ Hence, sampling from deterministic region occurs with higher probability



Example:

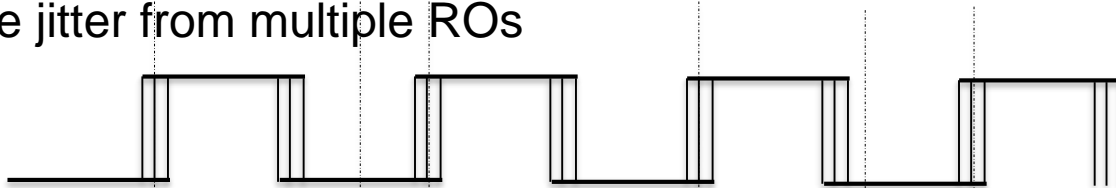
3% is non-deterministic

# Solving the practical problem of sampling jitter

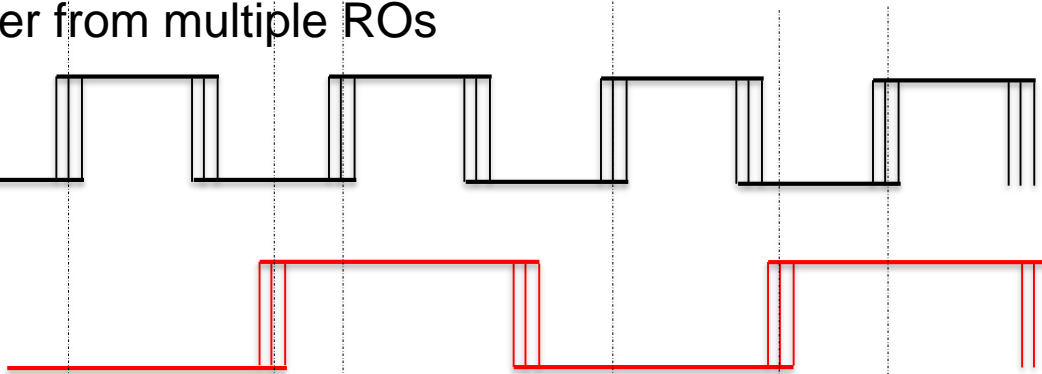
**Improve success rate:** Increase nondeterministic region compared to deterministic region

→ Combine jitter from multiple ROs

RO1



RO2



The % of nondeterministic region has increased

Combined jitter from two ROs  
(Only positive edge transitions are shown)

# Solving the practical problem of sampling jitter

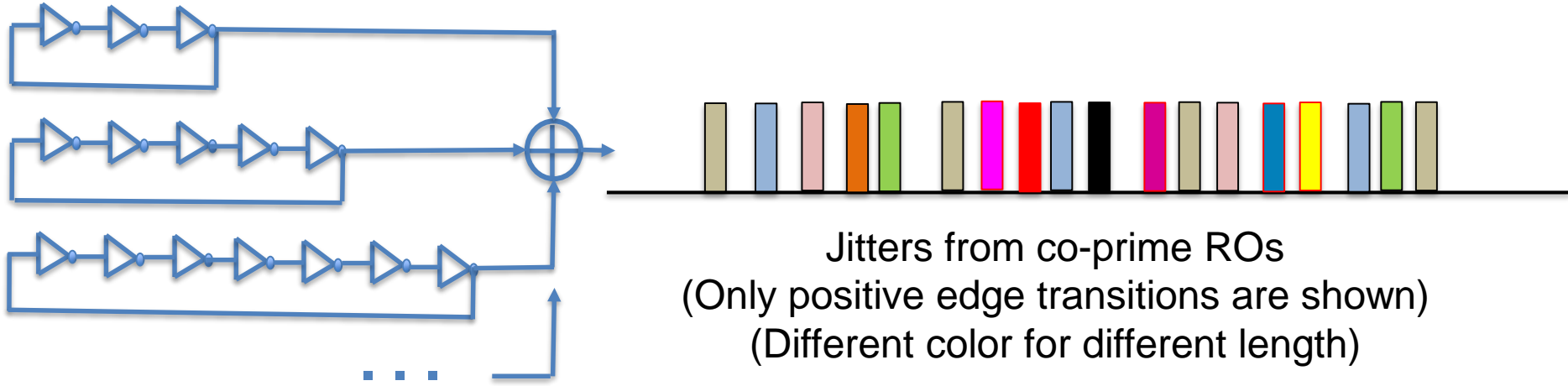
When combining multiple ROs, there are two main questions:

1. How many ROs should be combined?
2. What periods should they have?

# Sampling jitter using co-prime ROs

**Potential idea:** Use many ROs of *co-prime periods*, i.e., co-prime ring length.  
→ Hence, their 'jitters' will not overlap for long durations

Example: ROs of length 3, 5, 7, 11, ...



# Sampling jitter using co-prime ROs: Practical issues

- Assumption of non-overlapping jitter doesn't hold due to random phase-drift and RO-to-RO coupling effects → There will be overlaps more frequently
- Ring lengths increase dramatically → Area increases  
Ring lengths satisfying co-prime and odd: 1, 3, 5, 7, 11, 13, 17, 19, 23, ...

# Sampling jitter using co-prime ROs: Practical issues

- Assumption of non-overlapping jitter doesn't hold due to random phase-drift and RO-to-RO coupling effects → There will be overlaps more frequently
- Ring lengths increase dramatically → Area increases  
Ring lengths satisfying co-prime and odd: 1, 3, 5, 7, 11, 13, 17, 19, 23, ...

## Summary:

Enhancing jitter using Co-prime ROs is flawed and is not used in practice.



RO-based TRNGs use all ROs of the same length.

→ Due to random phase-drifts, their jitter-regions get spread across.

→ Implementation becomes easier.

**Question:** How many ROs should be used to ensure quality?

# The Urn Model [SMS07]

Let there are  $N$  urns.

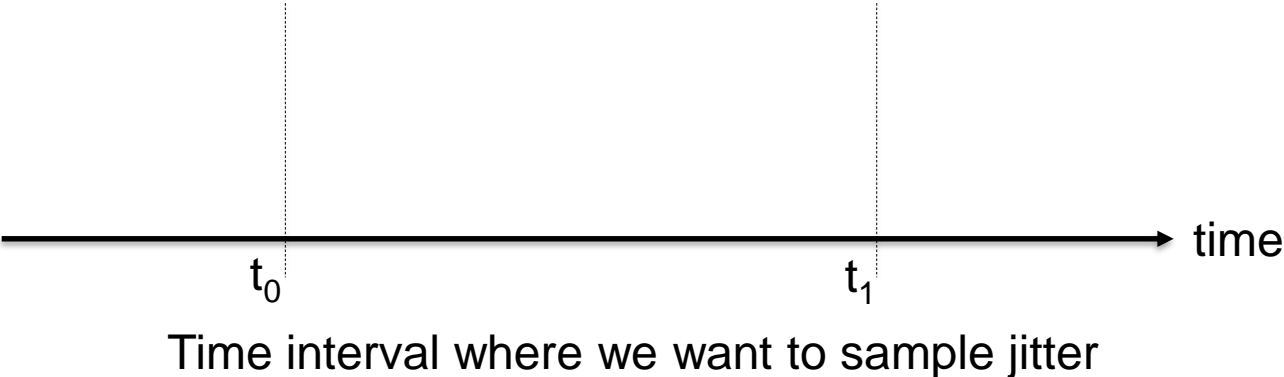
How many balls do we need to throw to fill all urns with high probability?



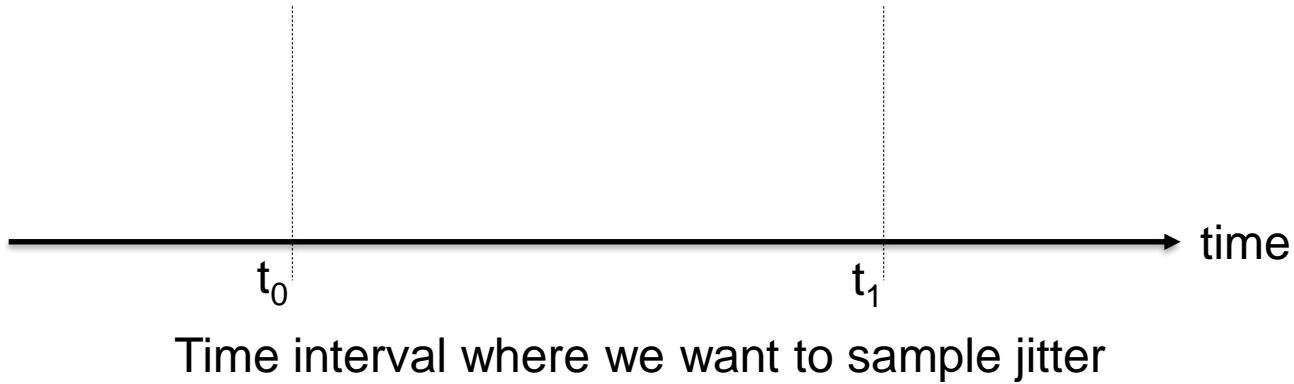
This problem is known as the ‘Coupon Collector Problem’.  
Expected number of balls:

$$r = \sum_{s=1}^N \frac{N}{s} = N \sum_{s=1}^N \frac{1}{s} \approx N \log N$$

# Apply Urn Model to calculate number of ROs



# Apply Urn Model to calculate number of ROs



The goal will be to fill this interval with 'transitions' (jitter) of RO outputs, such that any sampling point  $t$  falls within a 'transition' with high probability.

# Apply Urn Model to calculate number of ROs



The goal will be to fill this interval with 'transitions' (jitter) of RO outputs, such that any sampling point  $t$  falls within a 'transition' with high probability.

→ The interval is discretized by splitting it into  $N$  'urns' of equal width.

# Apply Urn Model to calculate number of ROs



The goal will be to fill this interval with ‘transitions’ (jitter) of RO outputs, such that any sampling point  $t$  falls within a ‘transition’ with high probability.

- The interval is discretized by splitting it into  $N$  ‘urns’ of equal width.
- Try to fill most of these urns with transition regions from ROs

Expected number of Ros: 
$$r = \sum_{s=1}^N \frac{N}{s} = N \sum_{s=1}^N \frac{1}{s} \approx N \log N$$

**How to decide the number ( $N$ ) of urns?**

## Summary so far

- Jitter of ring oscillator (RO) is unpredictable  
→ can be used to generate true random bits
- We need to combine jitter from many ROs to increase unpredictability
- ROs will be of equal ring-length
- Expected number of ROs can be derived using the Urn Model,

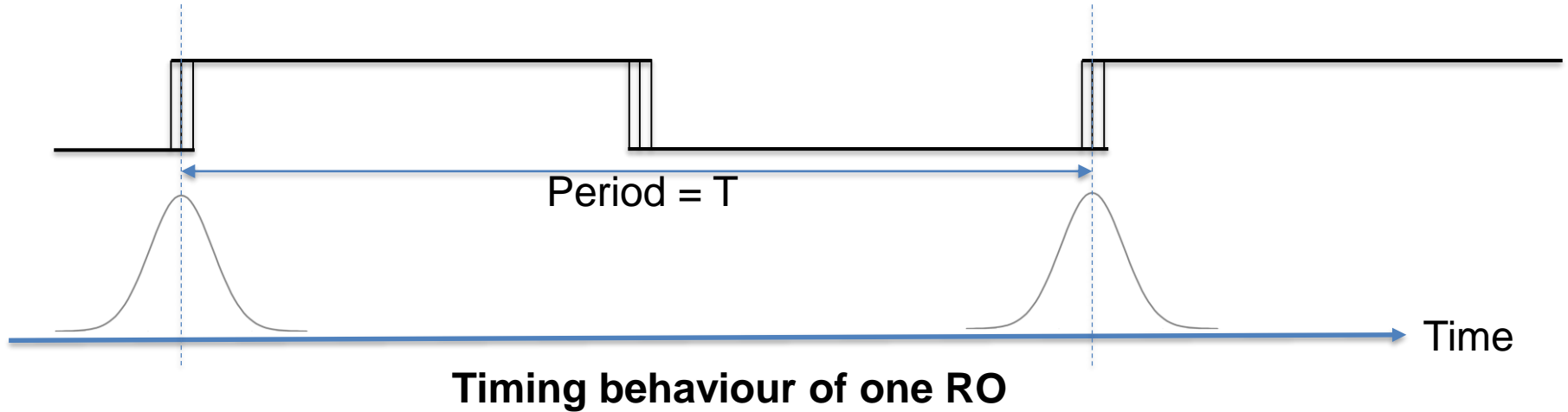
$$\text{Number of Ros: } r = \sum_{s=1}^N \frac{N}{s} = N \sum_{s=1}^N \frac{1}{s} \approx N \log N$$

where  $N$  is the number of urns.

- Next question: How to calculate the number of urns i.e.,  $N$ ?

# Calculating the number of urns

- As all ROs are of the same length, they have the same average period  $T$ .
- Jitter-width follows a Gaussian distribution with some standard deviation  $\sigma$ .



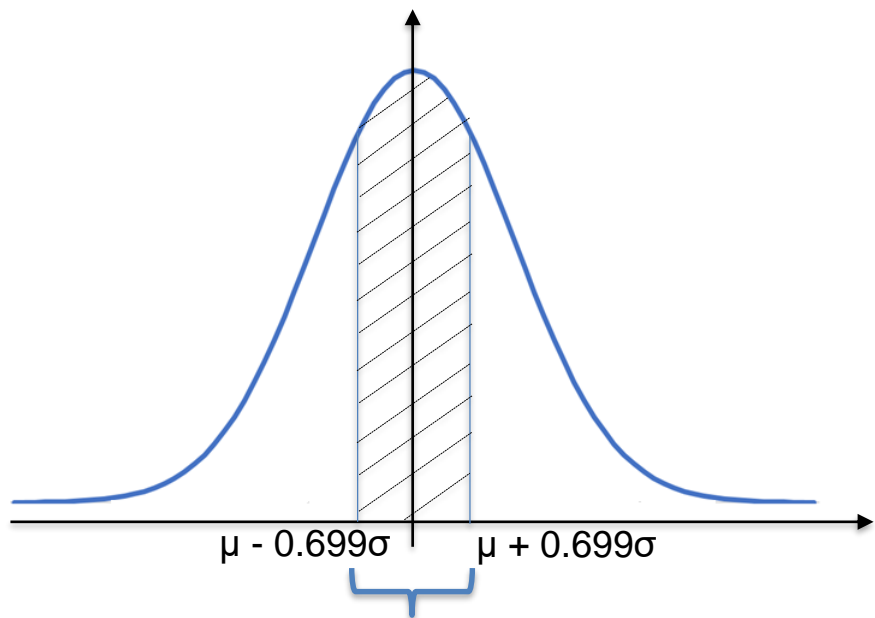
- We split each time-interval of width  $T$  into  $N$  urns.  
Hence, width of each urn =  $T/N$



# Calculating the number of urns

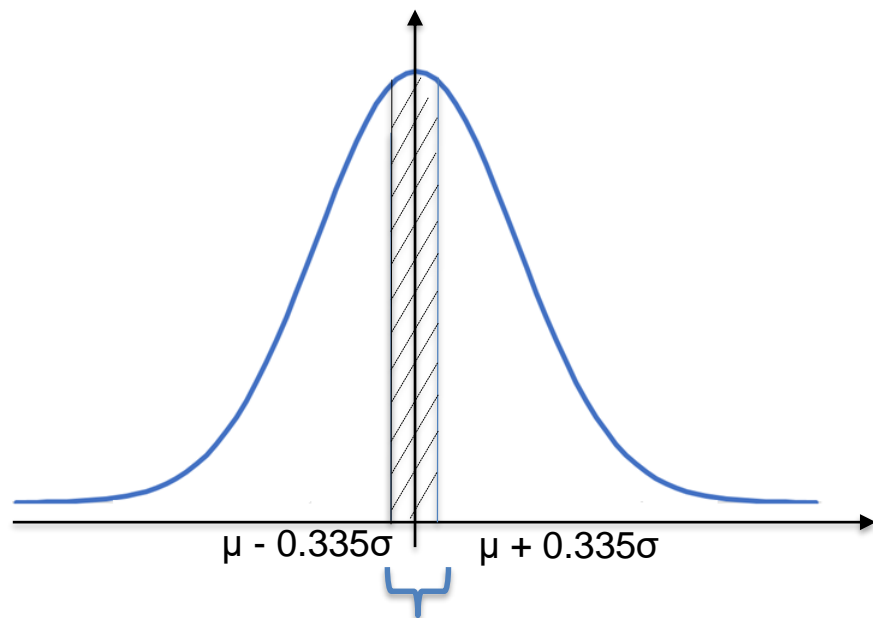
Let jitter be a Gaussian distribution of mean  $\mu$  and standard deviation  $\sigma$ .

Entropy in urn is related to urn-width.



Urn width =  $2 \times 0.699\sigma$

A sample here has entropy 0.80.



Urn width =  $2 \times 0.335\sigma$

A sample here has entropy 0.95.

# Calculating the number of urns

Entropy	Urn width
0.99	$2 \times 0.145\sigma$
0.97	$2 \times 0.258\sigma$
0.95	$2 \times 0.335\sigma$
0.90	$2 \times 0.479\sigma$
0.80	$2 \times 0.699\sigma$
0.50	$2 \times 1.229\sigma$

Narrower the urn,  
higher the entropy is.

Entropy vs urn-width

**Example:** If we want that the generated random numbers have entropy  $\sim 0.80$ , then

1. Urn-width  $w = 2 \times 0.699\sigma$
2. Thus, the **number of urns**  $N = T/w = T/1.398\sigma$

Typically  $\sigma$  is about 2% of the period  $T$ .

→ Hence to achieve entropy 0.80, we need  $N \approx 36$ . → Number of RO  $\approx 151$

## Summary so far

- Jitter of ring oscillator (RO) is unpredictable
  - can be used to generate true random bits
- We need to combine jitter from many ROs to increase unpredictability
- ROs will be of equal ring-length
- Expected number of ROs can be derived using the Urn Model,

$$\text{Number of ROs: } r = \sum_{s=1}^N \frac{N}{s} = N \sum_{s=1}^N \frac{1}{s} \approx N \log N$$

where  $N$  is the number of urns.

- #Urns  $N = T / (2 \times e \times \sigma)$  where factor  $e$  depends on desired entropy

**This gives us a formal model to estimate the number of ROs.**



So far, we have computed the expected number of ROs that we need to fill all the  $N$  urns.

→ Filling all the urns require a large number of ROs.

## ... relaxing the urn-filling condition



So far, we have computed the expected number of ROs that we need to fill all the  $N$  urns.

→ Filling all the urns require a large number of ROs.

The number of ROs can be reduced significantly if we aim for a lower filling rate  $f < 1$ .

E.g., with  $f = 0.7$  we have a 70% chance that the sampled value comes from jitter and 30% chance that it comes from deterministic values.

## Example: Expected number of ROs with filling rate $f = 0.7$

Let's assume that there are  $N=100$  urns.

With  $f = 0.7$  we expect :

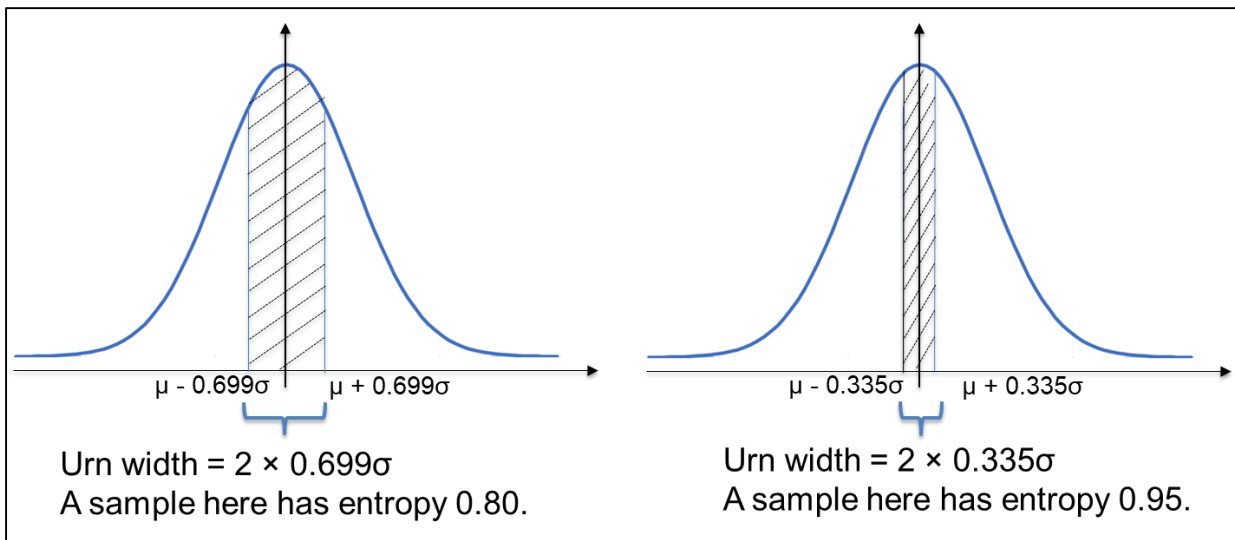
- 70 urns will contain jitter and
- the remaining 30 will contain deterministic values.

The expected number of ROs will be:

$$r = 100 \cdot \sum_{s=31}^{100} 1/s \approx 120$$

# Summary of the 'Urn model' (1)

1. You aim for a level of entropy and
2. and choose a proper width for the urns.



Jitter has a Gaussian distribution

## Summary of the 'Urn model' (2)

3. If possible, calculate the % of jitter for a RO on the target platform  
→ Measuring jitter requires a special circuit (not covered in this lecture)
4. Otherwise, choose the standard deviation of jitter  $\sigma$  to be 1% or 2% of the overall RO period.
5. Now, calculate  
#Urns  $N = T / (2 \times e \times \sigma)$  where factor  $e$  depends on desired entropy



## Summary of the 'Urn model' (3)

6. Expected number of ROs to fill all the urns (i.e.,  $f = 1$ ) with jitter

$$r = \sum_{s=1}^N \frac{N}{s} = N \sum_{s=1}^N \frac{1}{s} \approx N \log N$$

7. If we aim for a lower fill rate  $f < 1$  then the expected number of urns

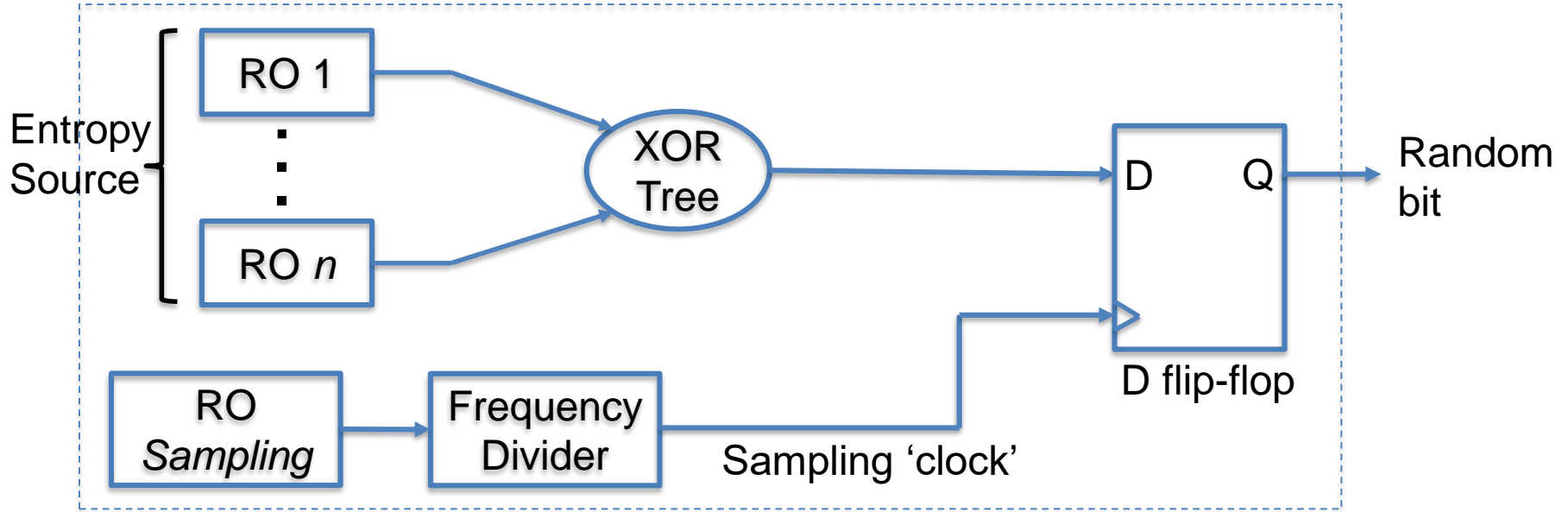
$$r = N \sum_{s=(1-f)N}^N \frac{1}{s}$$

8. In practice, you will need more ROs to have more 'confidence'.
9. With  $f < 1$  we reduce the number of ROs at the cost of quality. To compensate the loss in quality, we need to generate more random bits and then perform data compression (will be discussed next week).

## Implementation of RO-based TRNG

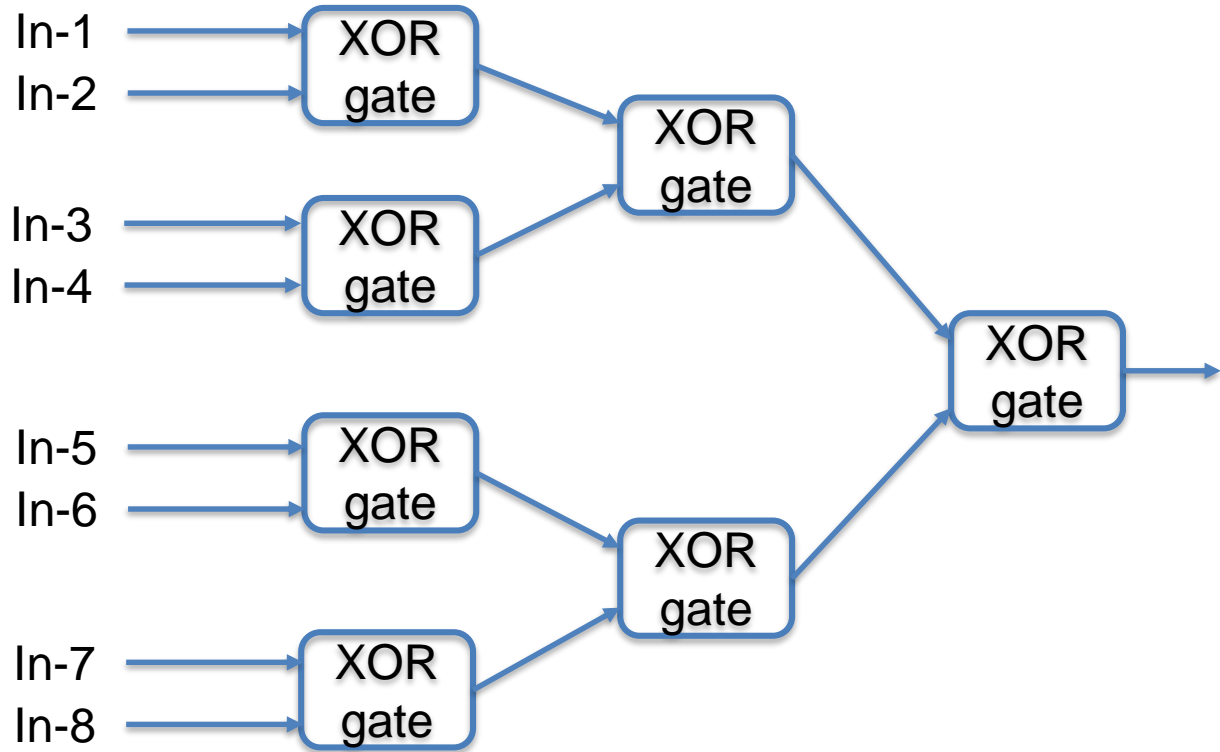
- RO-based TRNGs are popular and there are several ways of implementing them.
- We will cover only a few of them in this course.

# General structure of RO-TRNG



- The XOR-tree is a balanced arrangement of XOR gates with depth  $\log(n)$ . It accumulates transitions from all the  $n$  ROs.
- Sampling clock for the D-FF is generated from another RO and divided to obtain a much slower sampling frequency.

# Example of balanced XOR Tree

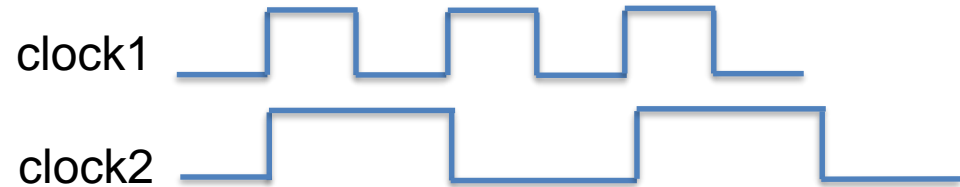
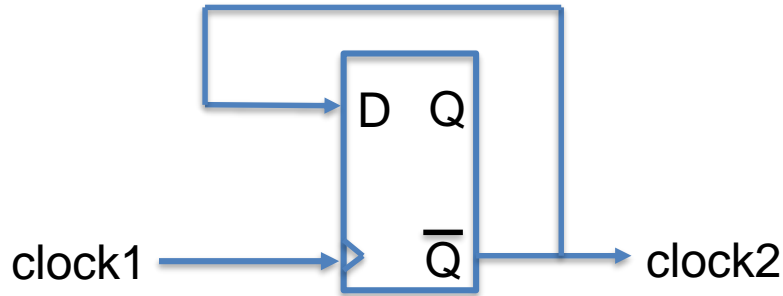


8 input XOR tree with depth 3

# Frequency divider

There are several ways of implementing a frequency divider.

- On FPGAs you have dedicated on-chip ‘Phased Locked Loop’ (PLL) IPs.
- To divide the clock by a power-of-2, then the easiest option is to use a cascade of D-FFs. Each D-FF divides its input clock by 2.



```
// Clock divider-by2 in Verilog  
always @ (posedge clock1)  
    clock2 <= ~clock2;
```

# RO-based TRNG of [SPV06] (1)

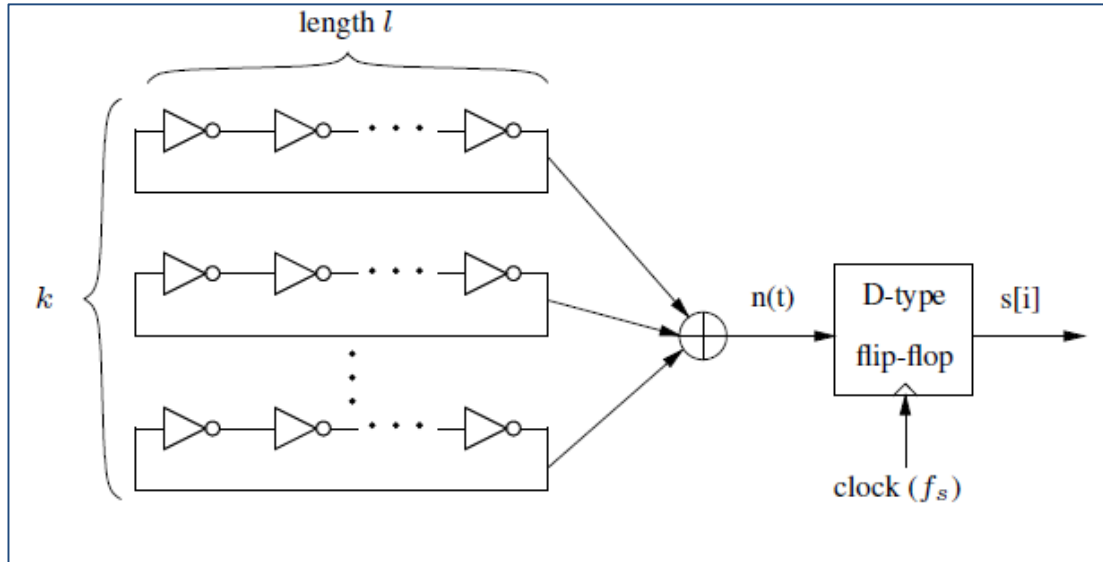


Image source [SPV06].

- Uses ROs of identical length.
- The  $k$  outputs are XOR-ed using a balanced XOR-tree to produce a single bit.
- The bit is sampled in a D-FF using a system clock of frequency  $f_s$ .

The authors performed various experimentations to determine  $l$  and  $k$ .

# RO-based TRNG of [SPV06] (2)

Number of ROs for different fill-rate ( $f$ ) and jitter-width

jitter/ period	fill rate $f$									
	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95
4%	45	53	59	70	79	94	107	133	158	231
2%	83	96	110	127	146	169	198	236	292	393
1%	158	182	210	241	277	320	374	445	548	733

Image source [SPV06].

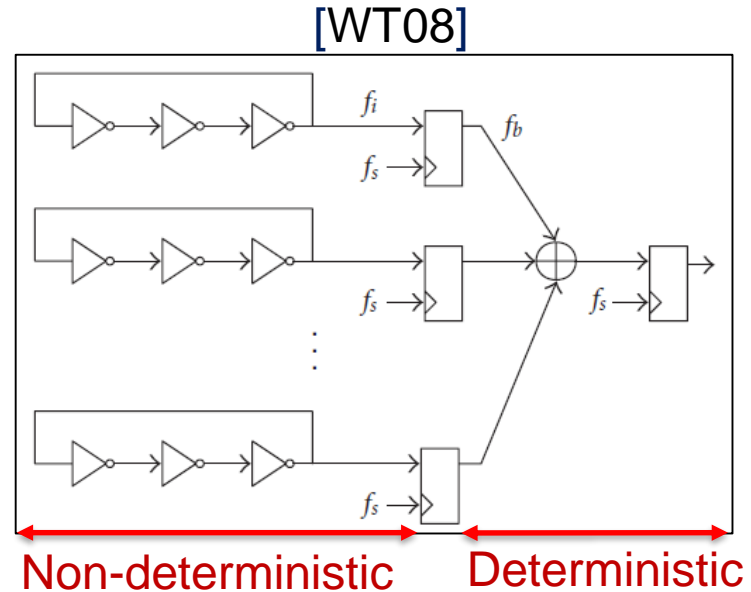
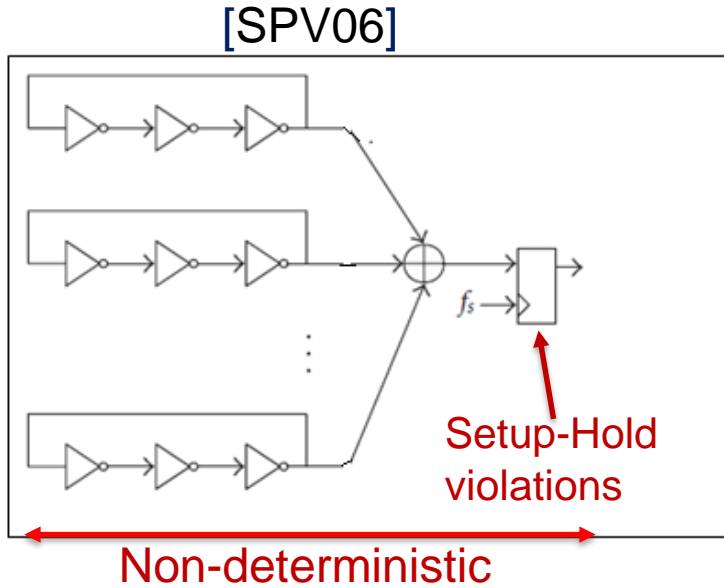
These two are implemented

The authors used all ROs of length  $l = 3$ .

Noise source	Number of ROs	Resources: #Slices on Xilinx Virtex 2 platform
Minimal	110	565
Robust	210	973

The D-FF is sampled at 40 MHz clock frequency.

# RO-based TRNG: Better jitter sampling [WT08]



In [SPV06] there are too many transitions at the input of the D-FF.

→ Causes setup and hold time violations for the D-FF.

Better approach [WT08]: Sample transitions of individual ROs first and then XOR them.

→ Uncertainty is captured in the first layer of D-FFs, and then accumulated in a deterministic way in the output D-FF.



# References

- [JK99] B. Jun, and P. Kocher. "The Intel Random Number Generator". White paper prepared for the Intel Corporation (1999).
- [SMS07] B. Sunar, W.J. Martin, and D.R. Stinson. "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks". IEEE Trans. on Comp., Vol. 56, No. 1, 2007.
- [Yang18] B. Yang, "True Random Number Generators for FPGAs," PhD thesis, KU Leuven, 154 pages, 2018. <https://www.esat.kuleuven.be/cosic/publications/thesis-307.pdf>
- [Rozic16] V. Rozic, "Circuit-Level Optimizations for Cryptography," PhD thesis, KU Leuven, 220 pages, 2016. <https://www.esat.kuleuven.be/cosic/publications/thesis-286.pdf>
- [SPV06] D. Schellekens, B. Preneel, I. Verbauwhede. "FPGA Vendor Agnostic True Random Number Generator". IEEE FPL 2006. DOI: 10.1109/FPL.2006.311206
- [WT08] K. Wold, and C.H. Tan. "Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings". Reconfigurable Computing and FPGAs, 2008.