Hardware-Oriented Security

Random Number Generation

Prof. Dr. Stefan Katzenbeisser

Lehrstuhl für Technische Informatik, FIM

Outline

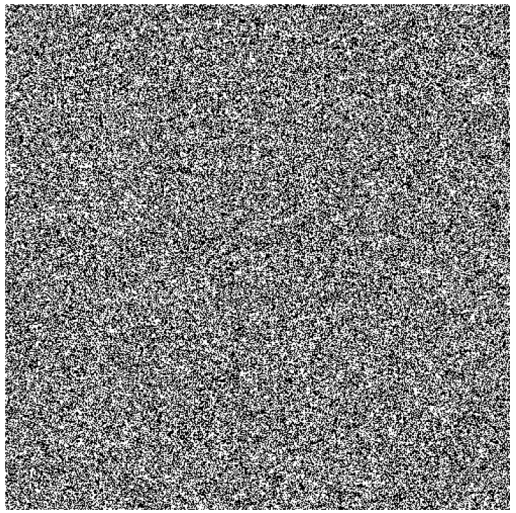
- What does randomness mean?
- What are random values used for?
- How can random values be acquired/generated?
- How can the quality of random values be determined?
  - What quality of random values is sufficient for the use in cryptography?

What is “randomness“?

Dictionary: without definite aim, direction, rule, or method
Wikipedia: lack of pattern or predictability in events

The impossibility to determine the precise and individual outcome of an event using a model, due to:

1. The nature of the model itself
2. Lack/Precision of model parameters

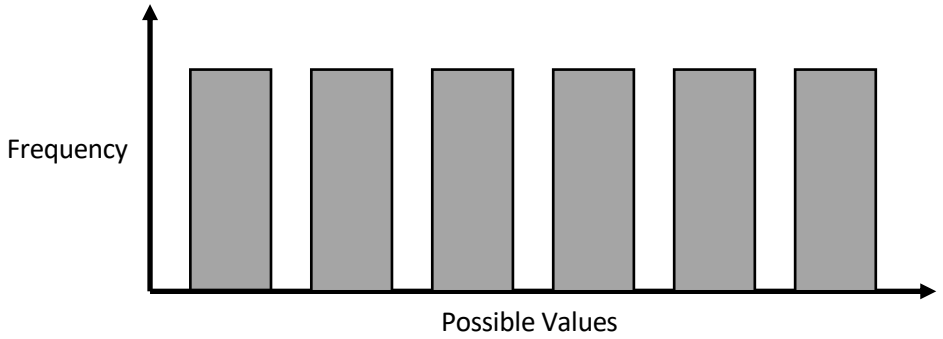


Randomness: Implications

When there is no possibility to make a prediction:

1. Individual options (numbers) must be equally probable
   → Uniformity
2. The occurrence of one option (number) must be independent of the occurrence of another option (number)
   → Independence

Each random number must be independently drawn from a uniform distribution:



Randomness: An algorithmic Definition

Randomness of a sequence is the **Kolmogorov complexity** of the sequence: size of smallest Turing machine that generates the sequence

Idea: When numbers in a sequence are uniform and independent,

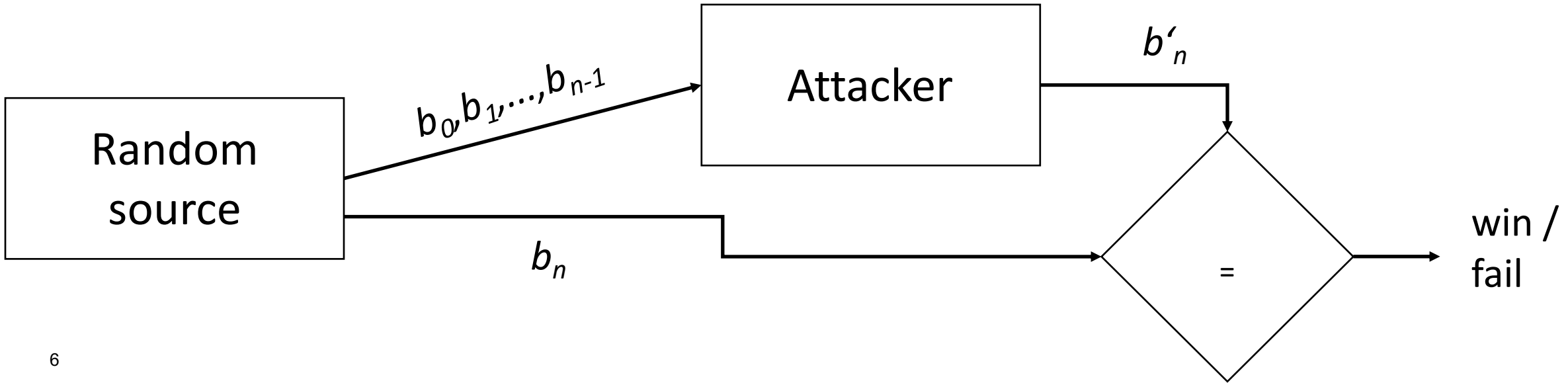
- an algorithm to output these numbers must be longer
- than the sequence itself → Incompressibility
- Intuitively: random strings canot be “produced” efficiently, but need to be “hardcoded” into the program

Problem: Measure is undecidable!

Randomness: a cryptographic definition

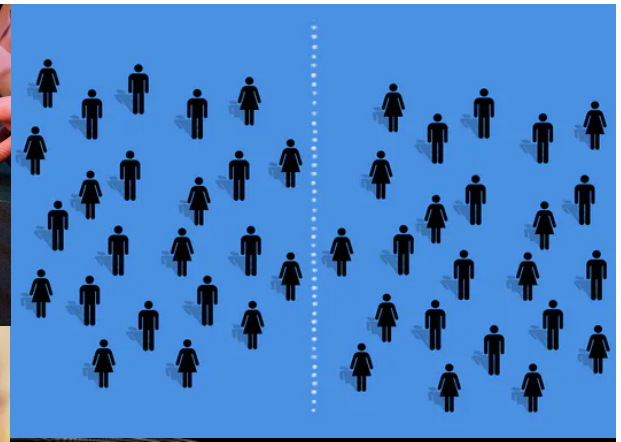
Key idea:

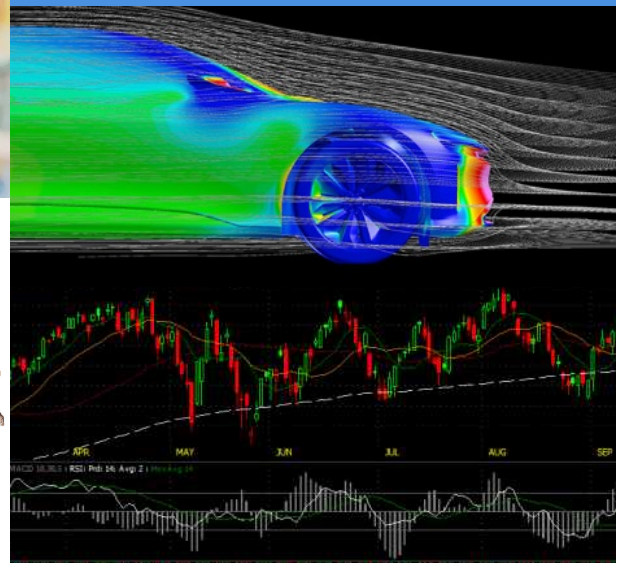
- Observe the output of a random bit stream for some time ...
- ... and try to predict the next bit
- Attacker wins, if he predicts bit with probability significantly greater than 1/2

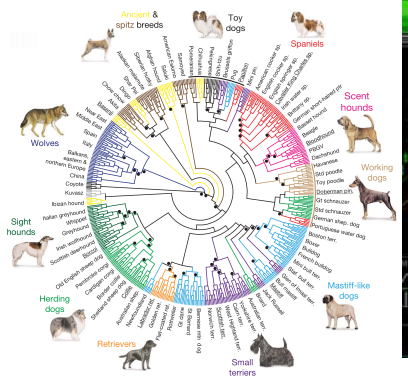


Applications of Random Numbers

- Opinion polls
- Quality control
- Simulations
- Evolution (mutations)
- Clinical trials
- Market forecasts
- Entertainment
  - Games (dice, cards, roulette)
  - Lotteries



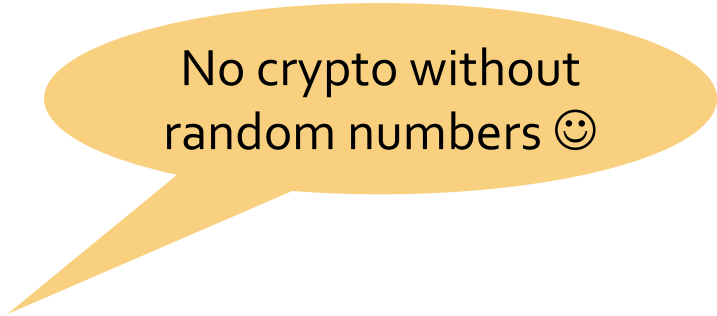




Applications of Random Numbers in IT Security

UNIVERSITÄT PASSAU

- Initial sequence number for network protocols (TCP,…)
- Session keys (TLS)
- Host keys (SSH)
- Salts for Hashes
- RSA prime factors
- Nonces (TLS, prevent replay attacks)
- Zero-knowledge protocols (random value for proving notion of discrete logarithm)
- Challenge-response protocols (challenge)
- Initializing vectors (AES-CBC,CTR,…)
- Encryption key in the one-time pad
- …



Example: RSA Key Generation

For creating RSA keys, two random primes of bit length k are required (n=pq):

1. Construction of an odd random number r:
   - Set last significant bit to 1
   - Set other bits randomly according to uniform distribution
2. Check whether r is a prime number by testing divisibility using prime numbers below some bound from a table of known prime numbers
3. If no divisor of r could be found → perform Miller-Rabin primality test
   - Requires more random numbers as witnesses for primality of r
   - For k = 1000 Bit, 3 witnesses guarantee the primality of r with probability $1-(1/2)^{80}$

This procedure is widely used and deployed

Security? (1)

Flaws in random number generators undermine crypto!

- Who would mess with a random number generator, right?

- Wrong! Introducing the case of Dual_EC_DRBG:
  - Dual_EC_DRBG is one of four RNGs proposals in NIST SP 800-90
  - Was only in the standard, because NSA wanted so;
    did not raise suspicion since „they are experts in making and breaking secret codes“
  - Small bias found in early 2006, but was resolved with workaround in appendix
  - Constants used to define elliptic curves in appendix, nobody knows where they came from
  - Researchers found in 2007 that a set of corresponding constants could be used to implement a backdoor, only 32 bytes of RNG output are enough to predict future outputs
  - Snowden documents confirmed that RSA quietly implemented DUAL_EC_DRBG as a default setting in their popular cryptographic library BSAFE
  - In late 2013 it was revealed that RSA in exchange received 10 Million USD from NSA

Security? (2)

CERT/CC is not aware of any demonstrated or reported attacks against applications of Dual_EC_DRBG.

Based on responses from vendors and publicly available information, the following vendors **do not** use DUAL_EC_DRBG in their products:

- Cisco
- Catbird Networks

The following vendors **do use** DUAL_EC_DRBG in their products, but **it is not enabled** by default:

- Blackberry
- Cummings Engineering
- Juniper (only in ScreenOS)
- Lancope
- McAfee
- Microsoft
- Mocana
- OpenSSL (only in the FIPS module)
- SafeLogic

The following vendors **do use** DUAL_EC_DRBG in their products, and **it is enabled** by default:

- RSA

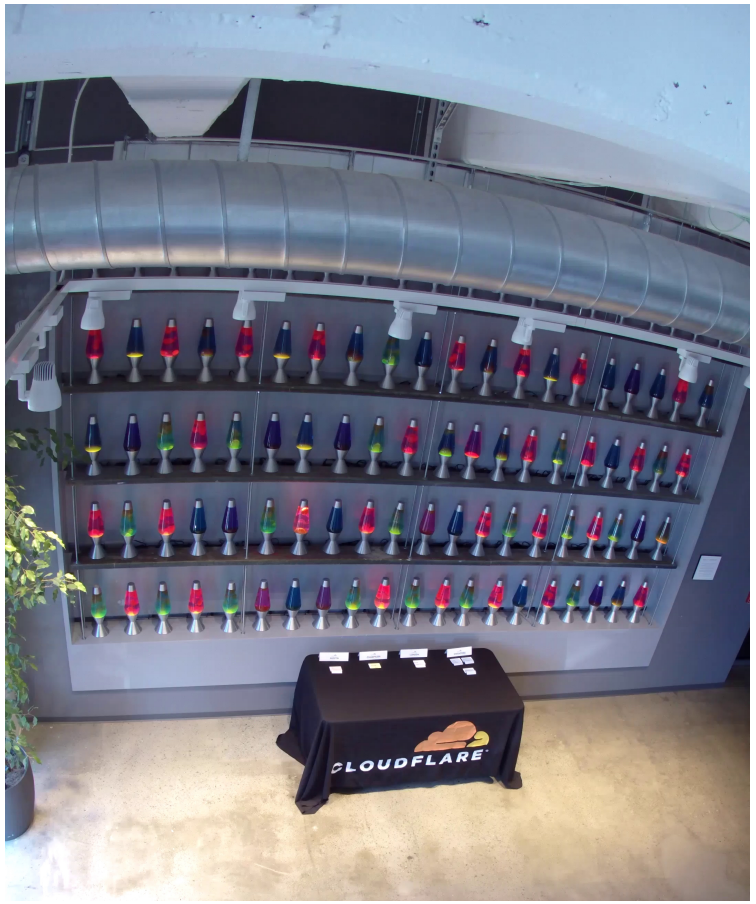
Source:
https://www.kb.cert.org/vuls/id/274923/

True vs. Pseudorandom numbers

- True Random Number Generators (TRNGs)
  - Usually based on physical processes, which may be microscopic or macroscopic
  - RNG's based on quantum properties are completely unpredictable
  - Unpredictable (in theory), but slow and possibly biased
  - Examples: Cards, coins, dice, roulette wheels, keyboard stroke timing, lava lamps, fishtank bubbles, radioactive decay, thermal noise, photoelectric effect

- Pseudorandom Number Generators (PRNGs)
  - Calculated by a computer through a deterministic (but complex) process
  - Disadvantage: not truly random
  - But: comparably fast, large quantities of random numbers can be generated

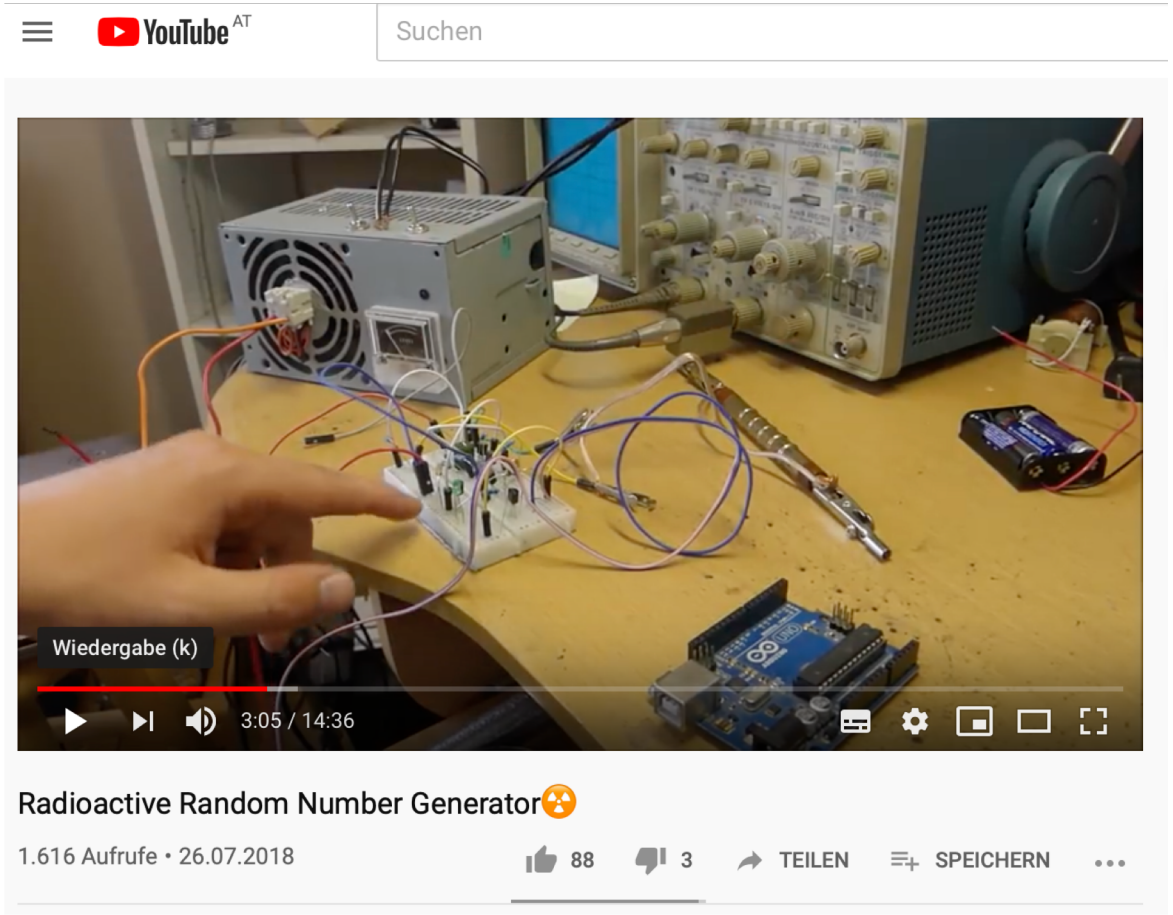
“Esoteric“ TRNGs



LavaRand

https://blog.cloudflare.com/randomness-101-lavarand-in-production/

Radioactive particles



https://www.youtube.com/watch?v=agvcduNRxKg&t=185s

Pseudo-Random Number Generators General Idea

Algorithmically produce numbers that simulate or imitate the ideal properties of random numbers.

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number - there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.

[PRNGs] … probably … can not be justified, but should merely be judged by their results. Some statistical study of the digits generated by a given recipe should be made, but exhaustive tests are impractical. If the digits work well on one problem, they seem usually to be successful with others of the same type.

John von Neumann

PRNG: Imperfections & Use

- Numbers calculated through a deterministic process, cannot be “random”
- Outputs must repeat eventually due to limited state space of algorithm (In contrast: TRNGs do not necessarily repeat)
- Given knowledge of the algorithm used to create the numbers and its initial state (seed), one can predict all subsequent “random” numbers

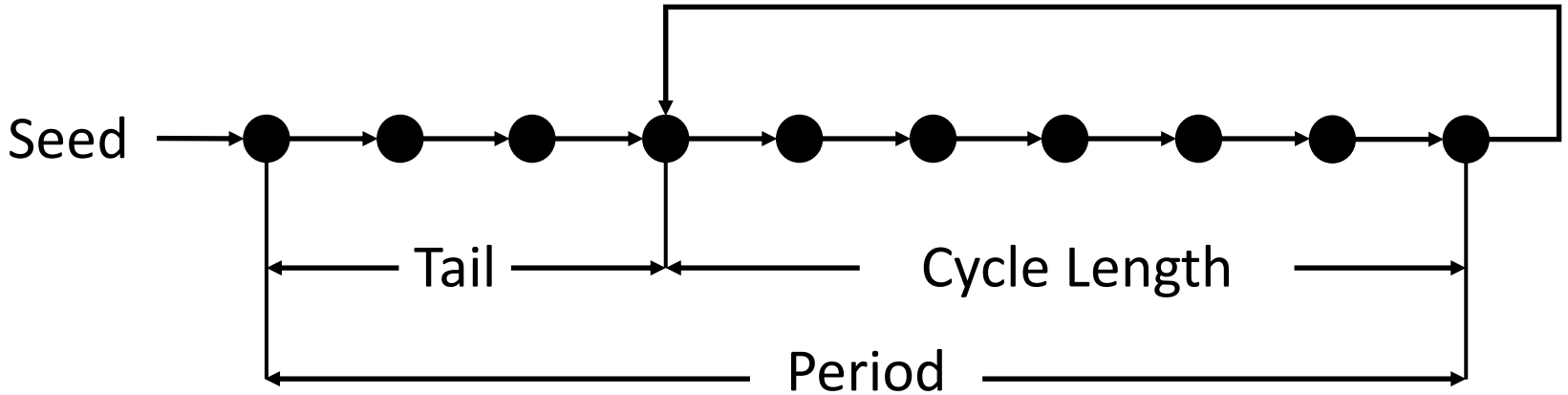
Typical use:

- Use TRNG to generate “*seed*”
- Expand seed to full random stream using PRNG

Hybrid approach: generate large quantities of random numbers fast at low cost!

PRNG: Structure

- Deterministic structure, generating required statistical properties
- Generator function is known
- Select generator function and seed such that period is maximal



PRNGs: Desirable properties

- Uncorrelated Sequences: The sequences should be serially uncorrelated (independence)
- Long Period: The generator should be of long period
- Uniformity: The sequence of random numbers should be uniform and unbiased
  - Equal fractions of random numbers should fall into equal ``areas'' in space
  - Example: if random numbers on [0,1) are to be generated, it would be poor practice if more than half to fall into [0, 0.1), presuming the sample size is sufficiently large
- Practicability:
  - Replicability: Verification and debugging
  - Efficiency: Numbers should be efficiently computable
  - Portability: Different architectures and operating systems

PRNGs: Constructions

Dedicated (and historic) constructions:

- Middle-Square Method
- Congruential Generators
- Mersenne Twister

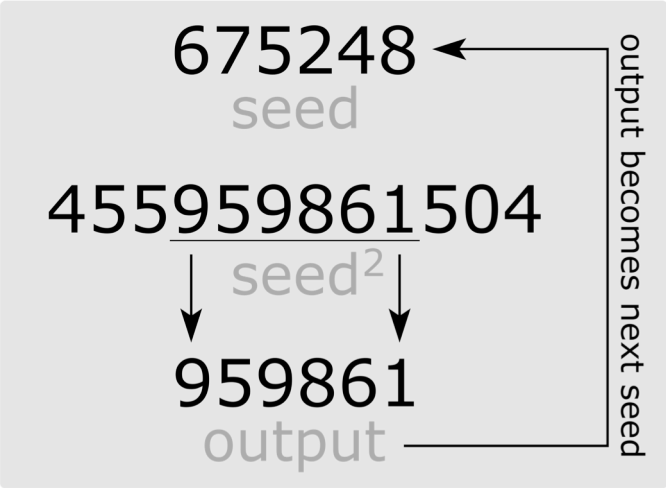
Cryptographic constructions:

- Blum-Blum-Shub generator
- Block ciphers (ANSI X9.17)

PRNGs: Middle Square Method (1)

- Proposed by von Neumann 1949 for ENIAC, since true random numbers took too long to read from punch cards
- Probably has been invented earlier by brother Edvin (13th century)

- Take an n digit number (n needs to be even or padded to be even, e.g. 4)
- Square it
- Take n digits from the resulting number, and so on
- Transfer to number range [0,1] by dividing by n



PRNGs: Middle Square Method (2)

Problems:

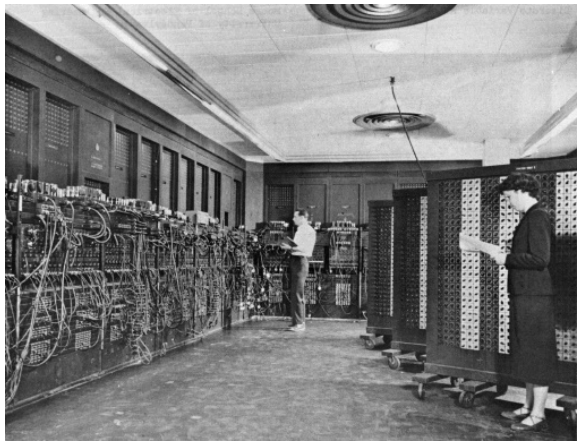
- Can converge to 0
- Can get stuck, quite often with zero but also with other numbers
- Can get into short loops
- Generates short periods of maximal length 8^n

Not favorable in practice!

PRNGs: Linear Congruential Generators (1)

$$x_{n+1} = ax_n + c \bmod m$$

- Start from initial seed x_0 and recursively compute random numbers
- Initially developed for ENIAC with *c=0,* but produced correlated sequences with low-order bits that were not random
- Selection of a,c,m,x_0 has *major* impact on cycle length!

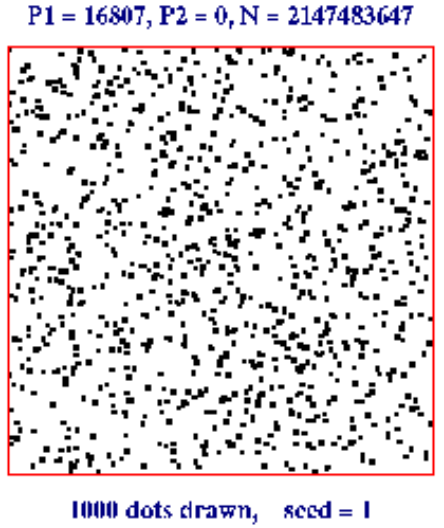


PRNGs: Linear Congruential Generators (2)

$$x_{n+1} = ax_n + c \bmod m$$

Popular Parameters:

- Park and Miller: a = 16807, c = 0, m = 2^{31}-1 = 2147483647, x_0 = 1
- ANSI C rand(): a = 1103515245, c = 12345, m = 2^{31}, x_0 = 12345



Plot of x_i vs. x_{i+1}

Congruential Generators: Example

Parameters:

x_0 =79, n = 100, a = 263, and c = 71

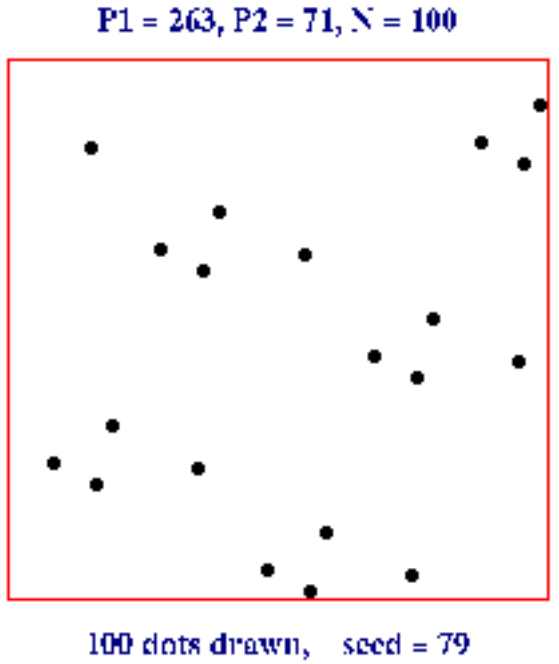
Computation:

x_1 = 79*263 + 71 (mod 100) = 20848 (mod 100) = 48

x_2 = 48*263 + 71 (mod 100) = 12695 (mod 100) = 95

x_3 = 95*263 + 71 (mod 100) = 25056 (mod 100) = 56

x_4 = 56*263 + 71 (mod 100) = 14799 (mod 100) = 99



Random sequence: 79, 48, 95, 56, 99, 8, 75, 96, 68, 36, 39, 28, 35, 76, 59, 88, 15, 16, 79, 48, 95

Choice of parameters (1)

- The modulus *m* should be large:
  - All x's are between *0* and *m-1*, the period can never be more than *m*
- For mod *m* computation to be efficient,
  - *m* should be a power of 2 $\Rightarrow$ mod *m* can be obtained by truncation
- If *b* is nonzero, the maximum possible period *m* is obtained if and only if:
  - Integers *m* and *b* are relatively prime, that is, have no common factors other than 1
  - Every prime number that is a factor of *m* is also a factor of *a-1*
  - If integer *m* is a multiple of 4, *a-1* should be a multiple of 4
- All conditions are met if $m=2^k$, $a = 4b + 1$, and *c* is odd (*c*, *b*, and *k* positive integers)

Choice of parameters (2)

- A generator that has the maximum possible period is called a *full-period generator*

$$x_n = (2^{34} + 1)x_{n-1} + 1 \mod 2^{35}$$
$$x_n = (2^{18} + 1)x_{n-1} + 1 \mod 2^{35}$$

- Lower autocorrelations between successive numbers are preferable
- Both generators have the same full period, but the first one has a correlation of 0.25 between x_{n-1} and x_n, whereas the second one has a negligible correlation of less than 2^{-18}

Choice of parameters (2)

- A generator that has the maximum possible period is called a *full-period generator*

$$x_n = (2^{34} + 1)x_{n-1} + 1 \mod 2^{35}$$
$$x_n = (2^{18} + 1)x_{n-1} + 1 \mod 2^{31}$$

- Lower autocorrelations between successiv
- Both generators have the same full perio
  0.25 between x_{n-1} and x_n, whereas the se
  less than 2^{-18}

Caution:
Linear Congruential Generators **can be predicted** given knowledge of a small number of random numbers produced.
Thus: **not** useful for crypto any longer!

Other Congruential Generators

- Linear congruential generators are only a special case of:

$$x_{n+1} = f(x_n, x_{n-1}, \ldots) \bmod m$$

- Quadratic generators:

$$f(x_i, x_{i-1}) = ax_i^2 + bx_{i-1} + c$$

- Higher-order linear generators:

$$f(x_i, x_{i-1}, \ldots) = a_1 x_i + a_2 x_{i-1} + \ldots + a_k x_{i-k}$$

- Special case, Fibonacci generator:

$$f(x_i, x_{i-1}) = x_i + x_{i-1}$$

Mersenne Twister (1)

- Invented by: Makoto Matsumoto and Takuji Nishimura (1997)

- Requires seeds Y_1 to Y_{624}
- Algorithm:

$$
\begin{aligned}
h &:= Y_{i-N} - Y_{i-N} \bmod 2^{31} + Y_{i-N+1} \bmod 2^{31} \\
Y_i &:= Y_{i-227} \oplus \lfloor h/2 \rfloor \oplus ((h \bmod 2) \cdot \mathtt{9908B0DF}_{\mathrm{hex}}) \\
x &:= Y_i \oplus \lfloor Y_i \,/\, 2^{11} \rfloor \\
y &:= x \oplus ((x \cdot 2^7) \wedge \mathtt{9D2C5680}_{\mathrm{hex}}) \\
z &:= y \oplus ((y \cdot 2^{15}) \wedge \mathtt{EFC60000}_{\mathrm{hex}}) \\
Z_i &:= z \oplus \lfloor z \,/\, 2^{18} \rfloor
\end{aligned}
$$

Mersenne Twister (2)

- Considered one of the “best” linear congruential generators
- Very fast, suitable for hardware implementation
- Negligible serial correlation

- Period is $2^{19937} - 1$ (a Mersenne prime)

- Again: suitable for Monte Carlo simulations, **but not for cryptography**

PRNGs for cryptographic applications
Required properties

Called "cryptographically secure PRNG" (CSPRNG)

- Output must indistinguishable from random
- **Uniformity:** At any point in the generation of the PRN sequence, the occurrence of a zero or a one is equally likely
- **Consistency:** Characteristics of sequence must not depend on the used seed
- **Unpredictability:** Next-bit test: Given a sequence of bits x_1, x_2, …, x_k, it is hard to "guess" the next bit x_{k+1}

- Theoretical result by Yao (1982): A sequence that passes the next-bit test, passes *all* other polynomial-time statistical tests for randomness
- **Forward security:** Given state of PRNG it is hard to compute previous output
  → mandatory requirement of the German evaluation guidance for PRNGs

CSPRNG: Blum-Blum-Shub (1)

- Invented by: Lenore Blum, Manuel Blum, and Michael Shub (1986)
- Based on difficulty of factoring, or finding square roots modulo $n = pq$

Initialization:

- p and q are primes $n=pq$
- Seed: random x relatively prime to n
- Initial state: $x_0 = x^2 \bmod n$

Random bit generation:

- i^{th} state = $x_i = (x_{i-1})^2 \bmod n$
- i^{th} bit = output least significant bit of x_i

CSPRNG: Blum-Blum-Shub (2)

- Based on public key algorithm
  - Security rests on difficulty of factoring n (security proof available!)
  - Cylce length rather large
- Parameter recommendations
  - *n* should be sufficiently large
  - *p,q* should be about same size but not too close *2 < p/q < 1000*
  - *p-1,p+1,q-1,q+1* each should contain a prime factor > $\sqrt[4]{n}$
- Unpredictable given any run of bits, passes next-bit test
- **Very slow**, since very large numbers must be used
  - Roughly one multiplication per random bit!
  - Good for key generation, Too slow for use in ciphers, poor choice for simulations

Fast CSPRNGs: Block Ciphers

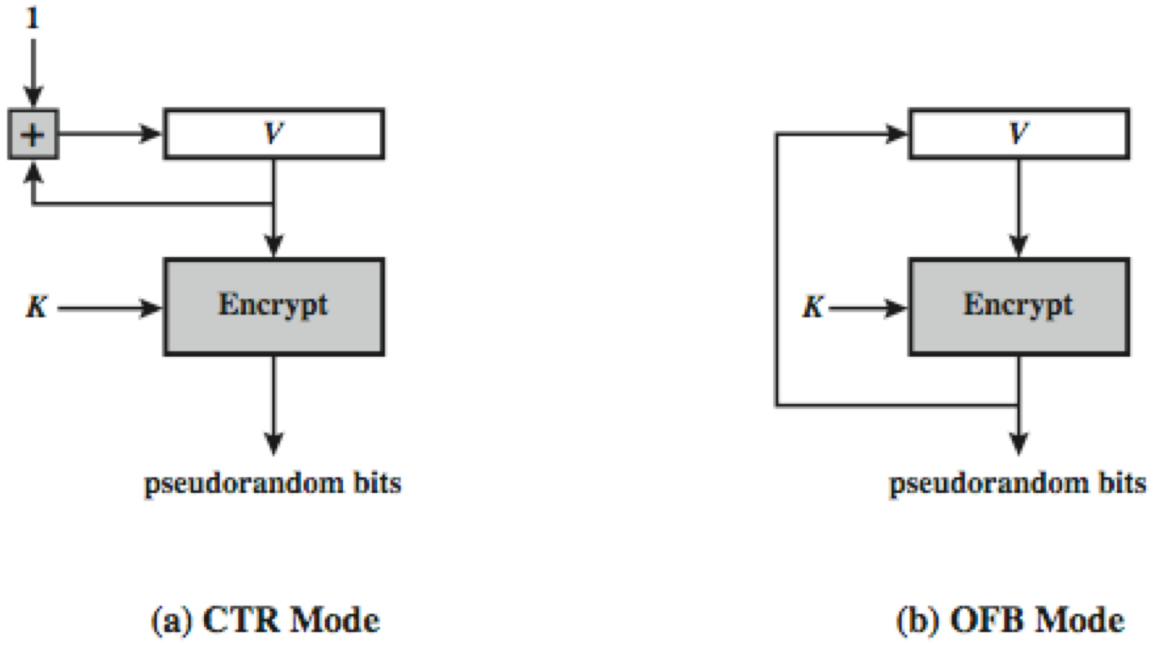
- For cryptographic applications, we can use a block cipher in an appropriate mode of operation to generate random numbers

- Counter Mode (CTR):

$$x_i = E_K[v_i]$$

- Output Feedback Mode (OFB):

$$x_i = E_K[x_{i-1}]$$



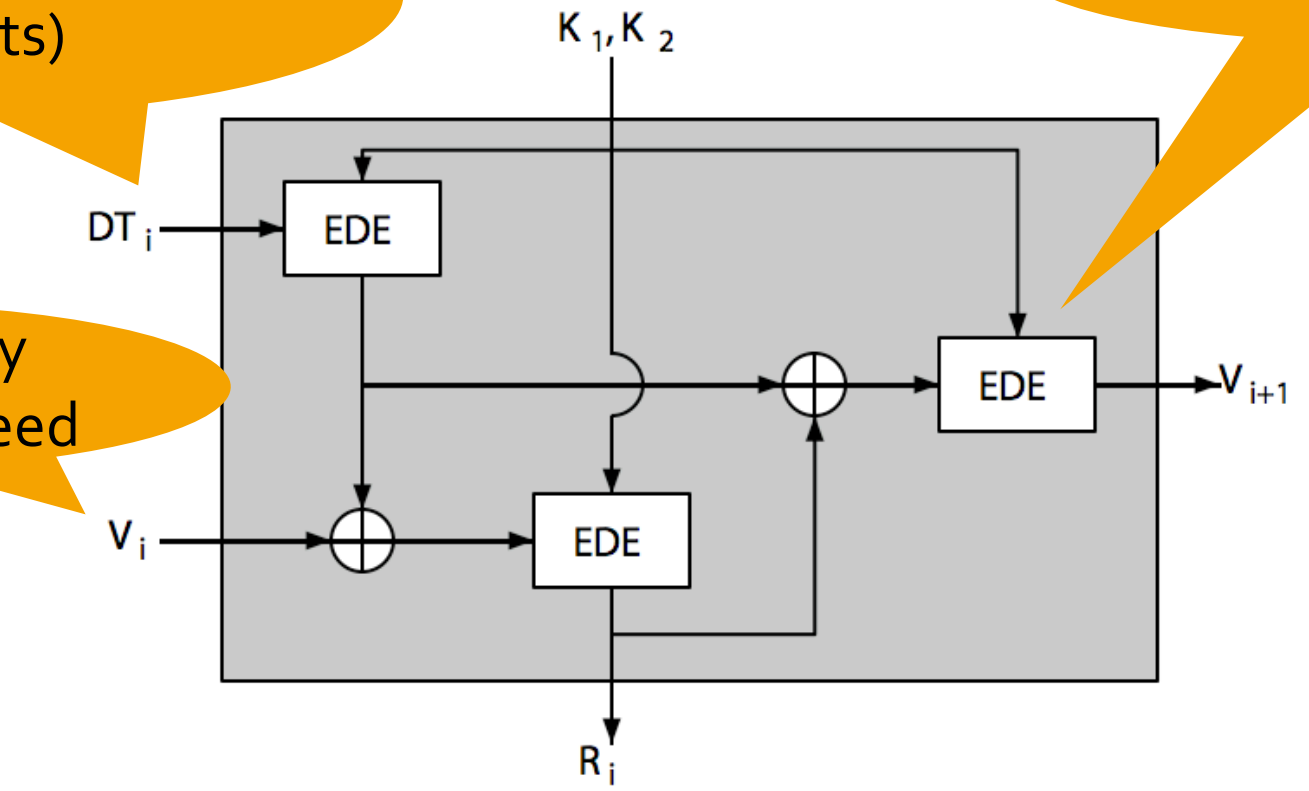
(a) CTR Mode (b) OFB Mode

Fast CSPRNGs: ANSI X9.17 PRG

Current date and time (64 bits)

Block Cipher: 3DES

State, initially initialized by seed



PRNG Attacks (1)

- **Direct Cryptanalytic Attack:** The attacker can directly distinguish between output of PRNG and random numbers (cryptanalyze the PRNG).
- **Input Based Attack:** The attacker is able to use *knowledge and control* of PRNG inputs to cryptanalyze the PRNG.
- **State Compromise Extension Attacks:** The attacker can guess state information due to an earlier breach of security.

Example: X9.17 PRNG: Vulnerable to *Input based attacks* and *state compromise extension attacks*

PRNG Attacks (2)
Direct Cryptanalytic Attack

- The attacker directly cryptanalyzes the PRNG
- Applicable to most PRNGs
- Not applicable when the attacker is not able to directly see the output of the PRNG.
  - For instance: A PRNG is used to generate triple-DES keys. Here the output of the PRNG is never directly seen by an attacker.

PRNG Attacks (3)
Input Based Attack

- An attacker uses knowledge or control of the inputs to cyptanalyze the PRNG output

- Types:
  - **Known Input:** If the inputs to the PRNG, designed to be difficult for a user to guess, turn out to be easily deducible (e.g., disk latency time). When the user is accessing a network disk, the attacker can observe the latency.
  - **Chosen input:** Practical against smartcards or applications that feed incoming messages (username/password etc.) to the PRNG as entropy samples.
  - **Replayed Input:** Similar to chosen input, except it requires less sophistication on the part of the attacker.

PRNG Attacks (4)
State Compromise Extension Attack

- Attacker attempts to extend the advantages of a temporary security breach
- These breaches can be:
  - Inadvertent leak
  - Previous cryptographic success
- This attack is successful if:
  - Attacker learns the internal state of the PRNG and is…
  - … either able to unknown PRNG outputs from before state was compromised …
  - … or able to compute subsequent outputs of the RNG
- These attacks usually succeed when the system is started in guessable state (due to lack of entropy)

PRNG Attacks (5)
State Compromise Extension Attack

Types:

- Backtracking attacks
  - Uses the compromise of PRNG state S to learn about all previous PRNG outputs.
- Permanent compromise attack
  - Once S has been compromised, all future and past outputs of the PRNG are vulnerable.
- Iterative guessing attacks
  - Uses the knowledge of state S that was compromised at time t and the intervening PRNG outputs to guess the state S’ at time t+Δ.
- Meet-in-the-middle attacks
  - Combination of iterative guessing and backtracking.

True Random Number Generators
Early Concepts

Dice:

- Forerunners were marked disks or bones, used for betting games
- Earliest six-sided dice date from around 2750 B.C. found in both northern Iraq and India
- Marked with pips, possibly because they pre-date numbering systems

Playing Cards:

- Used in China for gambling games as early as the 7th century
- Introduced to Europe in the late 1300's

Coins:

- Popular in ancient Rome, from whence they spread across Europe
- ~ 1900, English statistician Karl Pearson tossed a coin 24,000 times, resulting in 12,012 heads (f = 0.5005)

Spinning Wheels:

- Ancient Greeks spun a shield balanced on the point of a spear. The shield was marked into section, and players would bet on where the shield would stop.
- Evolved into fairground wheel of fortune, then into roulette

True Random Number Generators Natural Noise

UNIVERSITÄT PASSAU

- A True Random Number Generator (TRNG) uses a nondeterministic source to produce randomness
- Best source is *natural randomness* in real world
  - Chose a regular but random event and monitor
- Generally requires special hardware
  - Since relies on external physical quantities
  - Hardware start to emerge in new CPUs: Intel Ivy Bridge, Via Padlock
- Problems of bias or uneven distribution in measured signals
  - Requires processing
  - Best to only use a few noisiest bits from each sample

TRNGs: Desirable Features

Design:

- Simple design that allows rigorous analysis
- Compact and efficient design providing high thoughput per area and energy spent

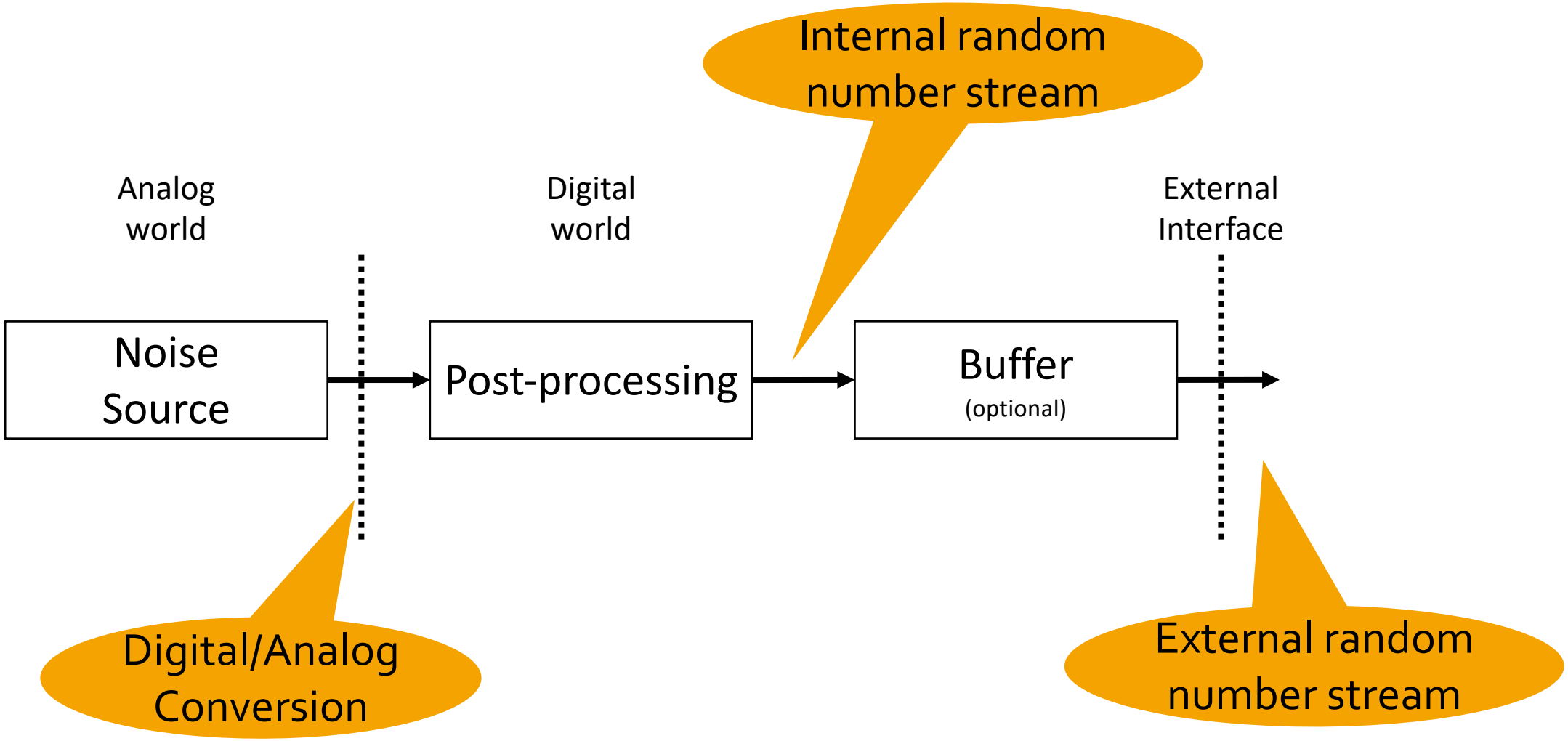
Evaluation:

- Mathematical justification for entropy collecting mechanism
- Empirical verification
- Validation of outputs using Test Suites (DIEHARD, NIST)

Implementation:

- Cheap silicon process
- Purely digital design process (no analog components such as amplifiers)

TRNGs: Generic Design



TRNGs: Physical Noise Sources

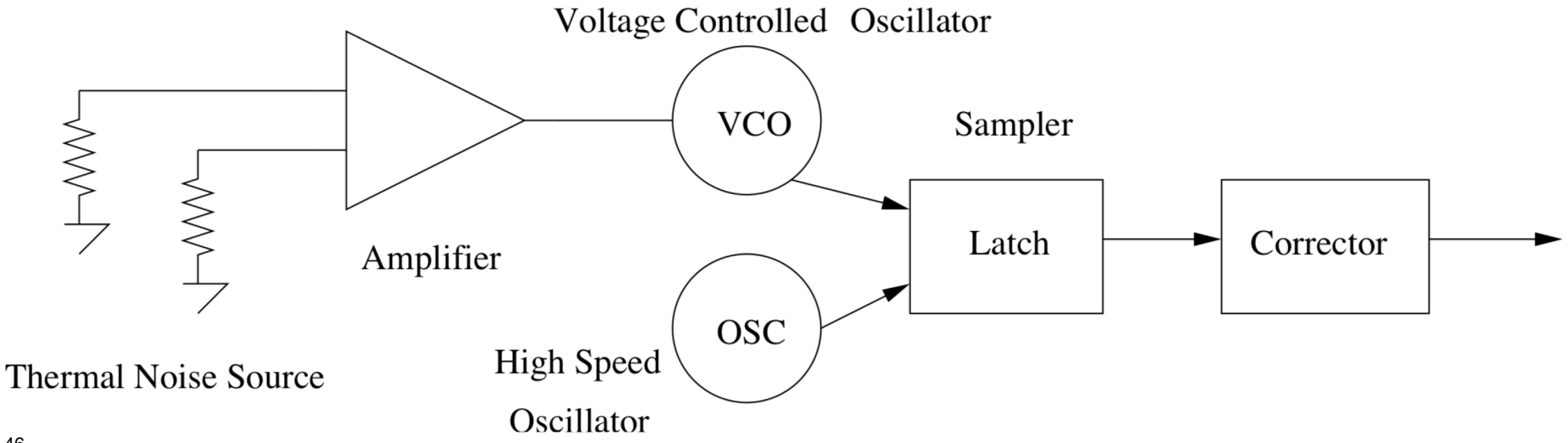
- Radioactive decay
- Radio frequency noise
- Audio noise
- Thermal noise in resistor or diode
- Leaky capacitors
- Mercury discharge tubes

TRNGs: Noise Sources in Computers

- Timing of keystrokes when a user enters a password
- Measurement of timing skew between two systems timers (HW/SW)
- Inter-interrupt timings (for some interrupts)
- Timings of memory accesses under artificially induced thrashing conditions
- Measurement of air turbulence due to the movement of hard drive heads
- Precise measurement of current leakage from a CPU or any other system component
- TPMs or other dedicated security hardware:
  - Ring oscillators
  - Metastable components (flip flops, memory cells), …

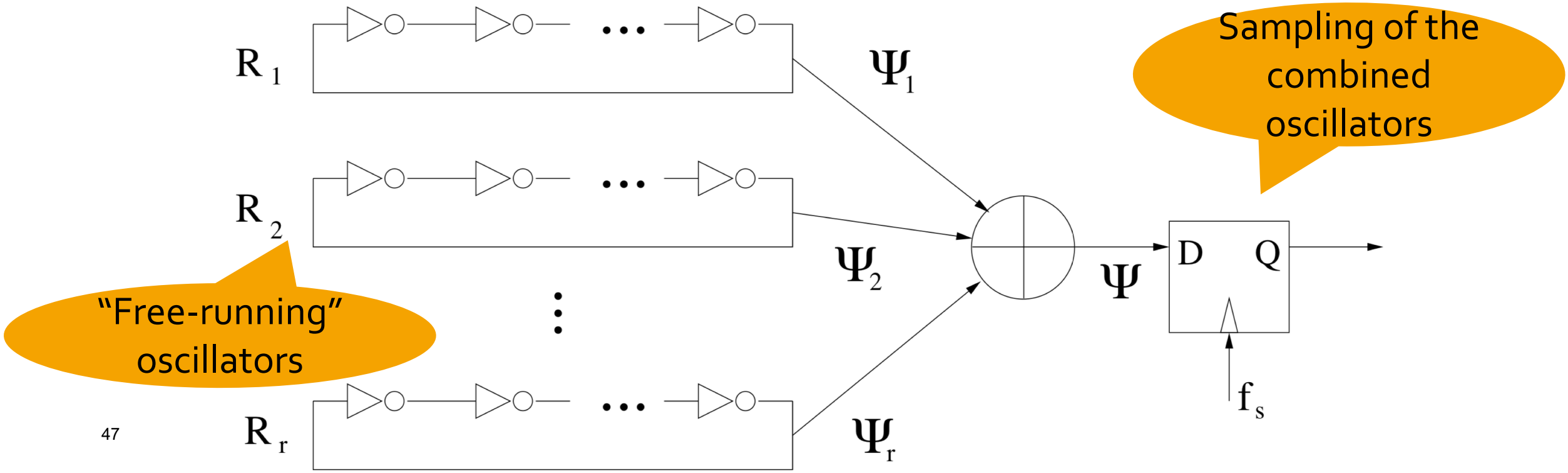
Example TRNG Constructions (1) Intel TRNG

- Thermal noise is amplified to drive an oscillator, which is sampled by another oscillator
- Output is post-processed



Example TRNG Constructions (2) Ring Oscillators

- Jitter in oscillators is a good source of randomness.
- Oscillation frequency slightly different (random) from device to device.



TRNG Postprocessing

- Physical randomness sources used by TRNGs have imperfections:
  - Implementation errors
  - Parasitic, non-ideal components
  - Sampling can introduce correlation

- Solution:
  - Entropy distillation: used to convert biased random bits to unbiased random bits
  - Examples: von Neumann, parity-based, hash function-based

TRNG: Postprocessing
Von Neumann Correction

- Introduced 1951 to remove bias in random sequences
- Approach:
  - Collect two random bits
  - Discard if they are identical
  - Otherwise, use first bit

- Idea: If the bits in the sequence are independent, but not with the same probability , i.e. the probability of “1” is p and the probability of “0” is q, with p ≠ q, then the two symbols “01” and “10” have the same probability p · q, while the symbols “11” and “00”, with different probabilities p^2 and q^2, are discarded.
- Sequence gets shorter, by 0.25 if numbers are random, much more for long sequences of 1s and 0s

TRNG: Postprocessing
Von Neumann Correction

- Introduced 1951 to remove bias in random sequences
- Approach:
  - Collect two random bits
  - Discard if they are identical
  - Otherwise, use first bit

Note: correction **always** discards randomness. The more "non-uniform" the source, the more raw random bits are required!

- Idea: If the bits in the sequence [illegible] the same probability , i.e. the probability of [illegible] of "0" is q, with p ≠ q, then the two symbols "01" and "[illegible]" occur with probability p · q, while the symbols "11" and "00", with diffe[illegible]t probabilities p^2 and q^2, are discarded.
- Sequence gets shorter, by 0.25 if numbers are random, much more for long sequences of 1s and 0s

TRNG Postprocessing
Parity Based Method

- Input stream divided into n-bit chunks
- For each chunk a parity bit is computed, the chunk is then thrown away
- Parity bit: n-bit XOR of all the bits in the chunk

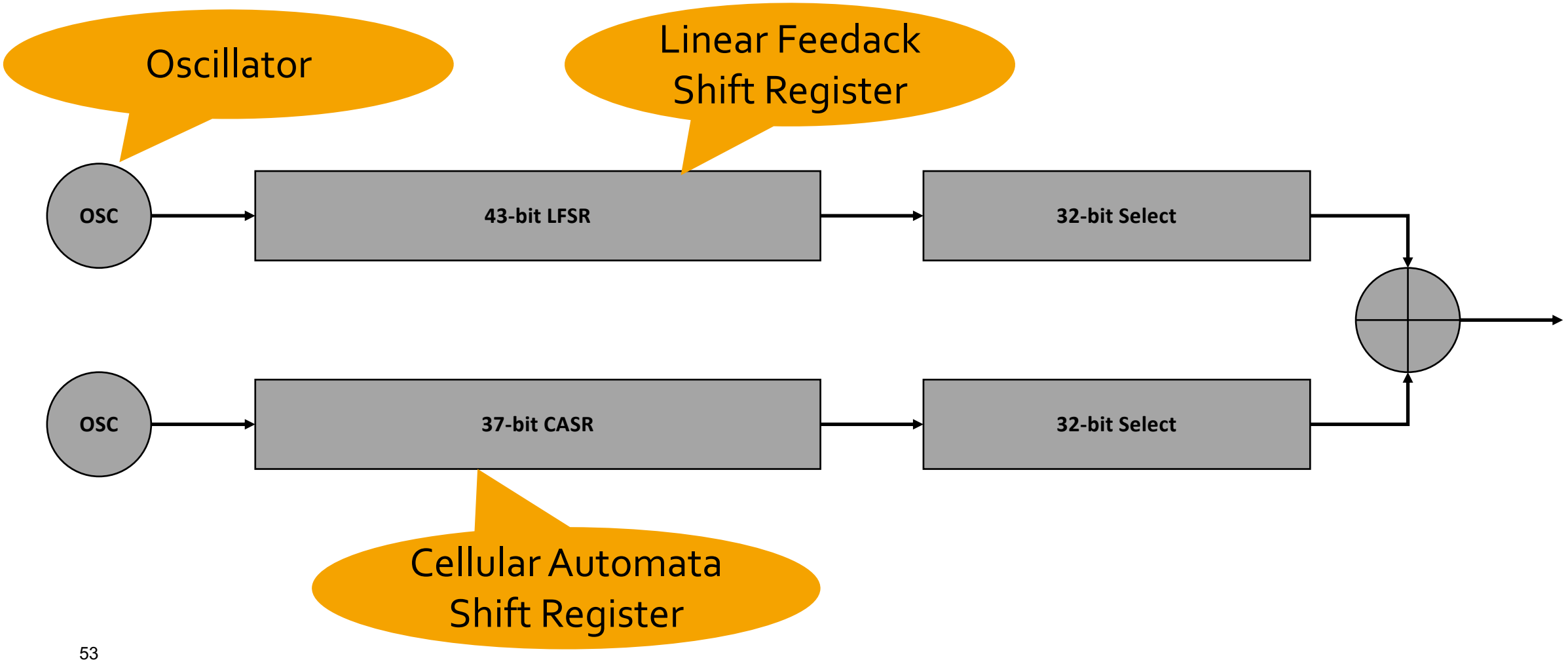
- Idea:
  - n-bit XOR reshapes correlation and increases entropy
  - Truth table of n-bit XOR has 2^{n-1} zeros and 2^{n-1} ones

- Downside: heavy data reduction 1/n

TRNG Postprocessing
Universal Hashing

- Hash functions turn arbitrary sized data into a number of fixed length
  - Function chops and mixes (substitutes and transposes) data to create fingerprint
  - Good hash function → few collisions, difficult to find to data with same hash
  - Fully algebraic, no memory of previously hashed data
  - Commonly used: SHA-1 (160 bit) or similar
  - Decimation rate can be controlled
- Problem: Computationally heavy
  - 160 Bit state space = 5 * 32 bit words
  - Each stage basic operations (add, rot, substitute, non-linear ops), but 80 stages
- If “perfect” hash function is used, imperfections can be removed completely

Advanced Design: Tkacik

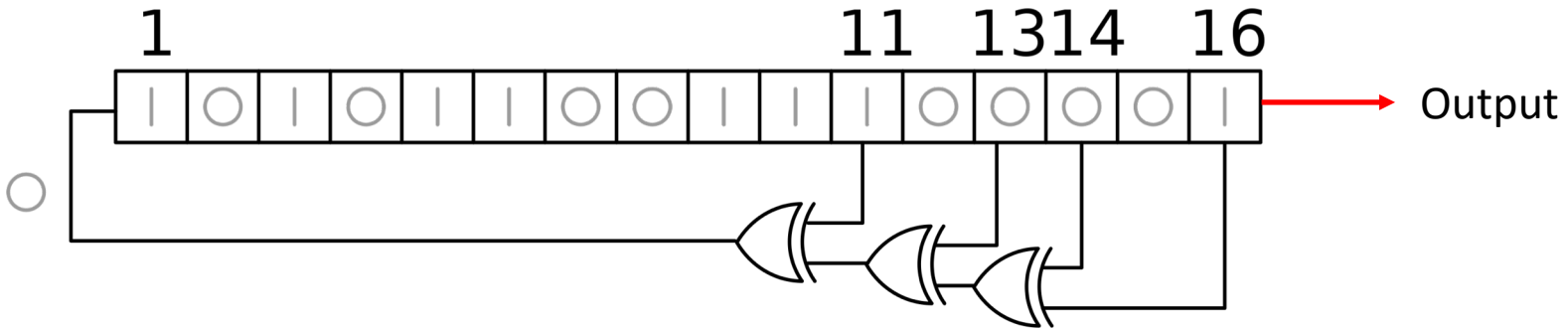


LFSR: Linear Feedback Shift Register

LFSRs can be predicted! Must be combined in a RNG!

- Storage of a fixed number of bits
- Some bits are XORed together to generate new input
- State of register is “shifted”: new value is added, oldest value is sent to output
- Properties determined by ”characteristic polynomial”, e.g.

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$



Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1								

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0						

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0	0					

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0	0	0				

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0	0	0	1			

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0	0	0	1	0		

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90
State 0	0	1	0	1	1	0	1	0
State 1		0	0	0	1	0	0	

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule	90	90	90	150	90	90	90	90	
State 0	0	1	0	1	1	0	1	0	0
State 1		0	0	0	1	0	0	1	

Cellular Automata Shift Register

- Various “rules“ to modify data in a register, based on neighborhood

Rule90 : $01011010_2 = 90_2$

Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	0	1	0	1	1	0	1	0

Rule150 : $10010110_2 = 150_2$

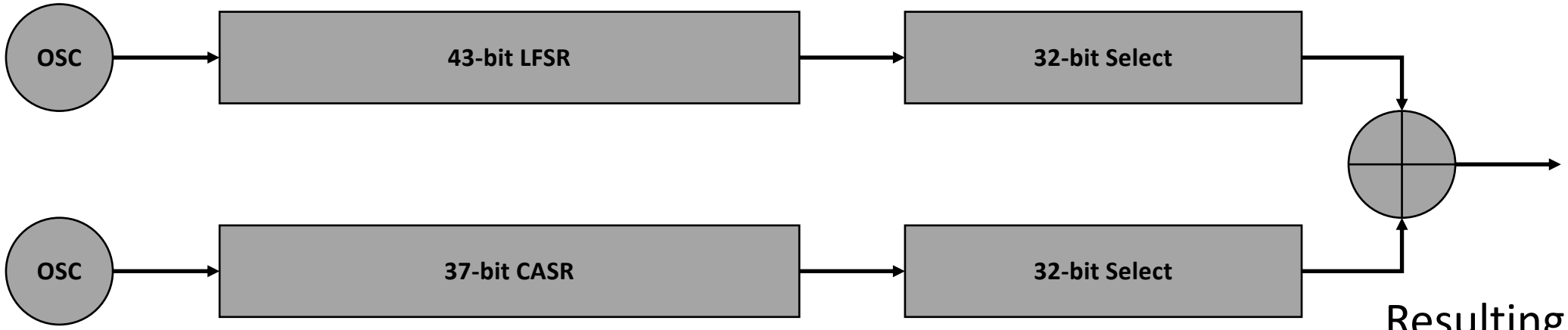
Number	7	6	5	4	3	2	1	0
Neighbour-hood	111	110	101	100	011	010	001	000
Rule Result	1	0	0	1	0	1	1	0

Rule		90	90	90	150	90	90	90	90
State 0	0	0	1	0	1	1	0	1	0
State 1		1	0	0	0	1	0	0	1

Advanced Design: Tkacik

LFSR:

- Feedback polynomial: $x^{43} + x^{41} + x^{20} + x + 1$
- Cycle length of $2^{43} - 1$



CASR:

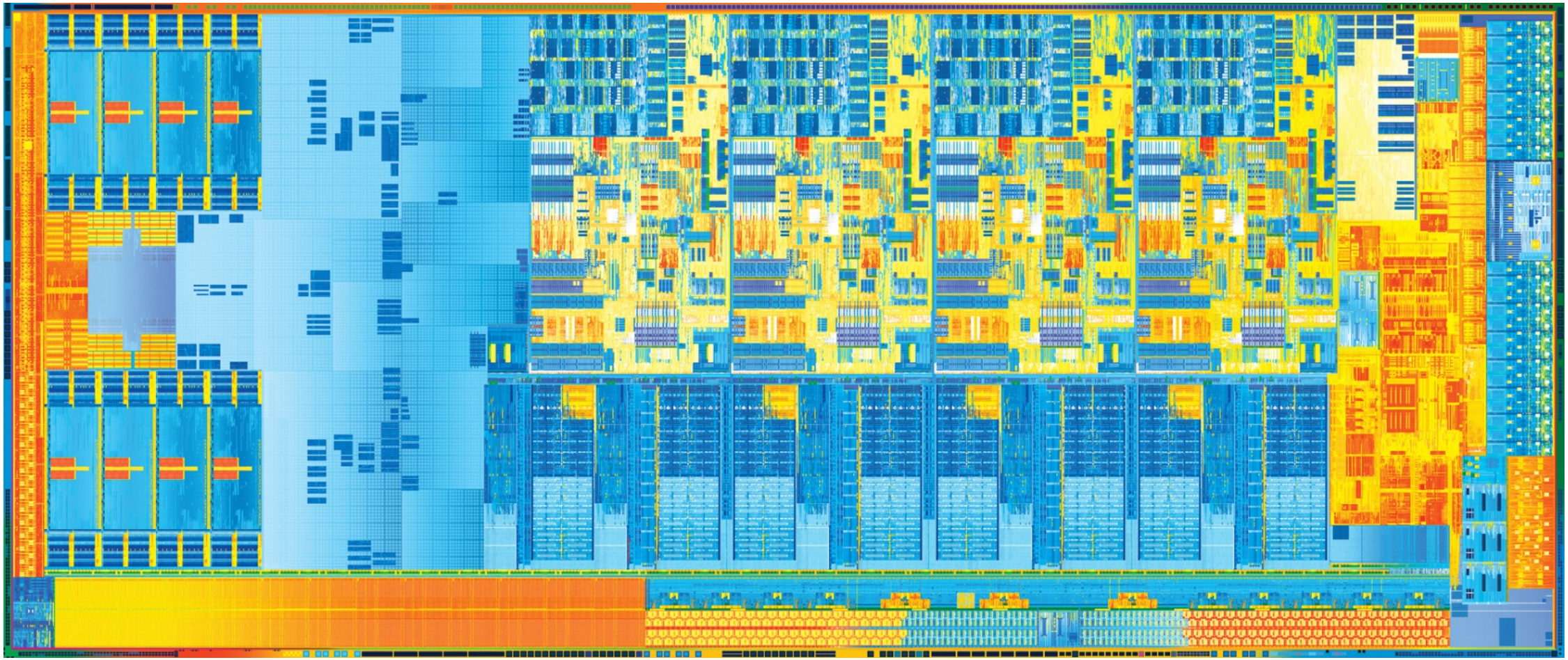
- CA150 at cell site 28, and CA90 at all other cell sites
- Cycle length of $2^{37} - 1$

Resulting cycle length:
$2^{80} - 2^{43} - 2^{37} + 1$

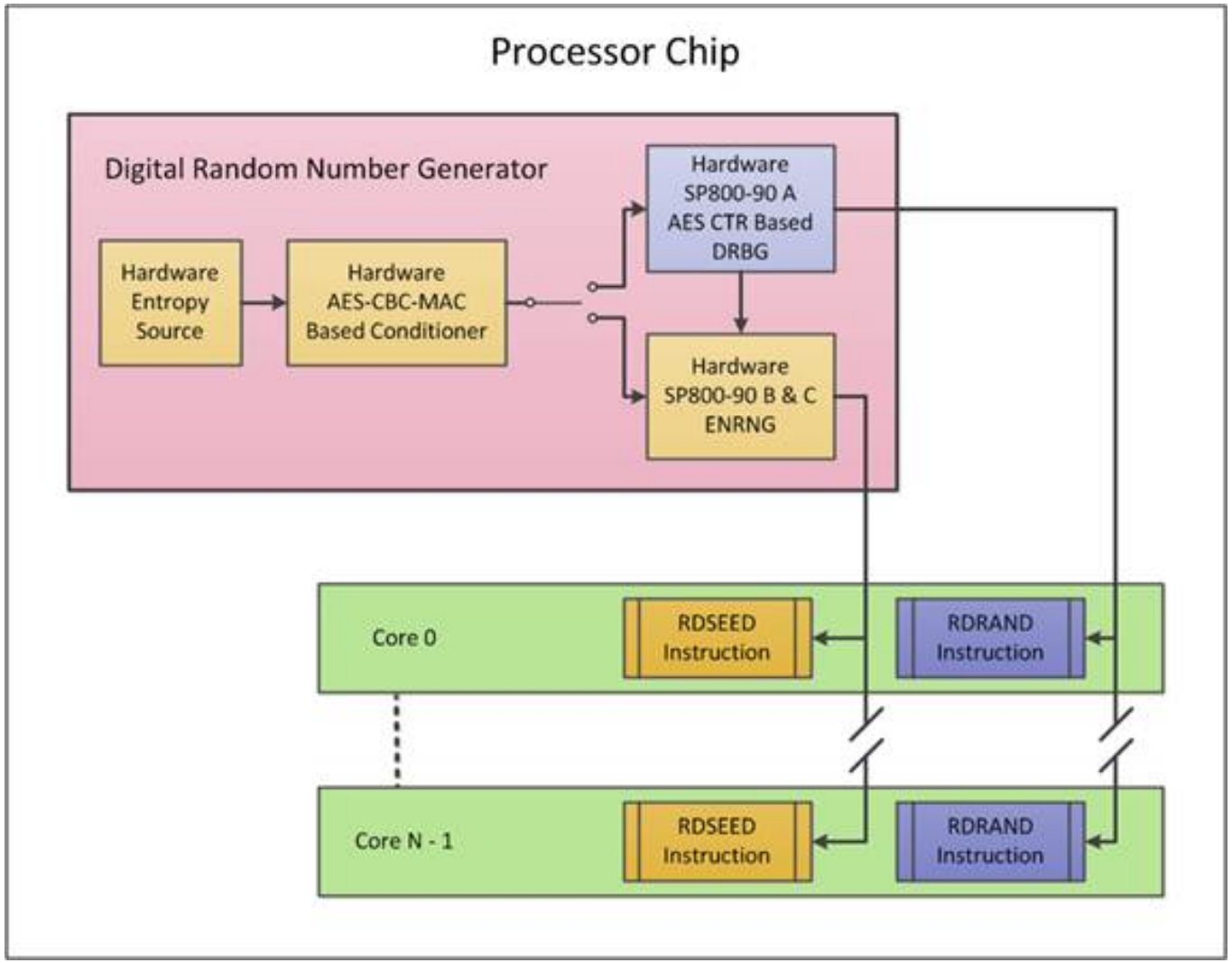
Advanced Design: Tkacik Properties

- Outputs 32 bits at a time
- Randomness comes from jitter of the two free running oscillators
- Passes major tests (DIEHARD, NIST)
- Entropy of only 2 oscillators is limited
- Linear components make design susceptible to attacks
- Can be made robust by lowering output rate significantly (1/60000)

TRNG: Real Stuff



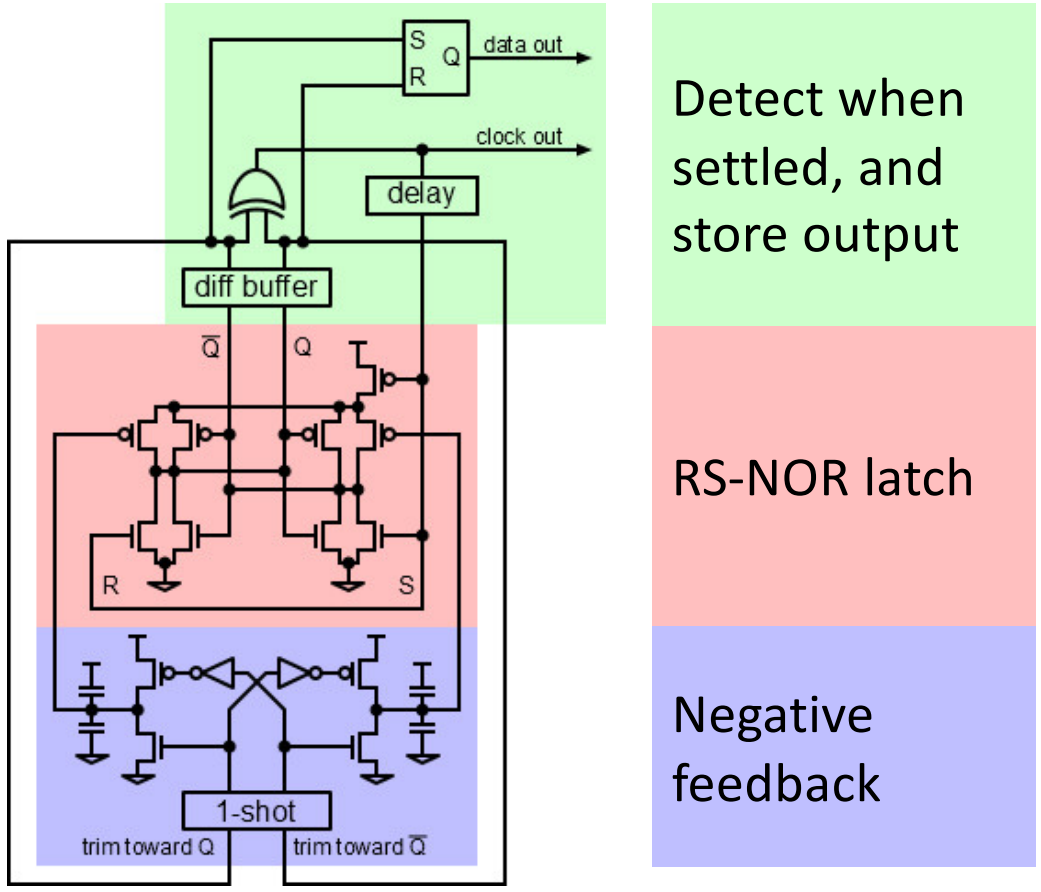
Intel Ivy Bridge General Design



Two versions to generate random streams:

- Deterministic Random Number Generator (AES in CTR mode)
- Enhanced Non-deterministic Random Number Generator for direct generation of seeds.

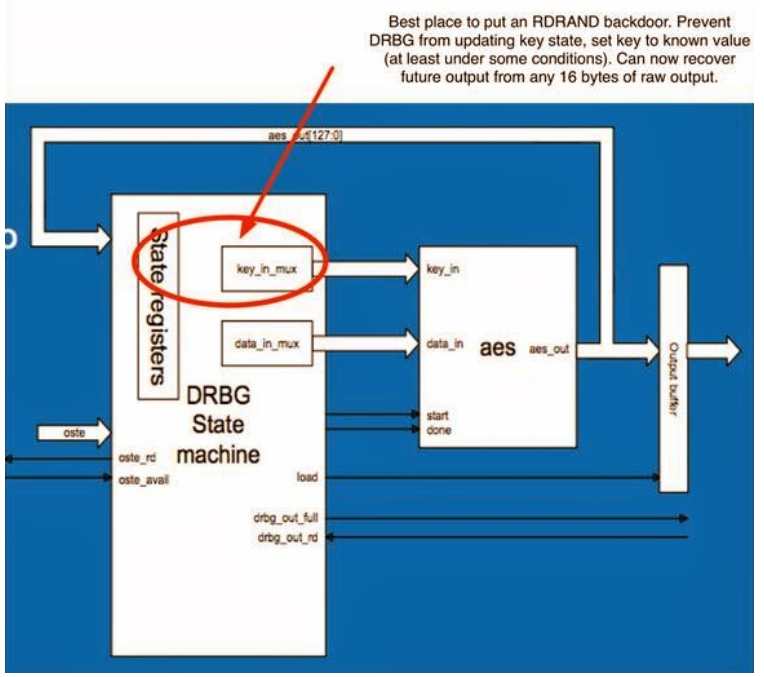
Intel Ivy Bridge
Source of Randomness



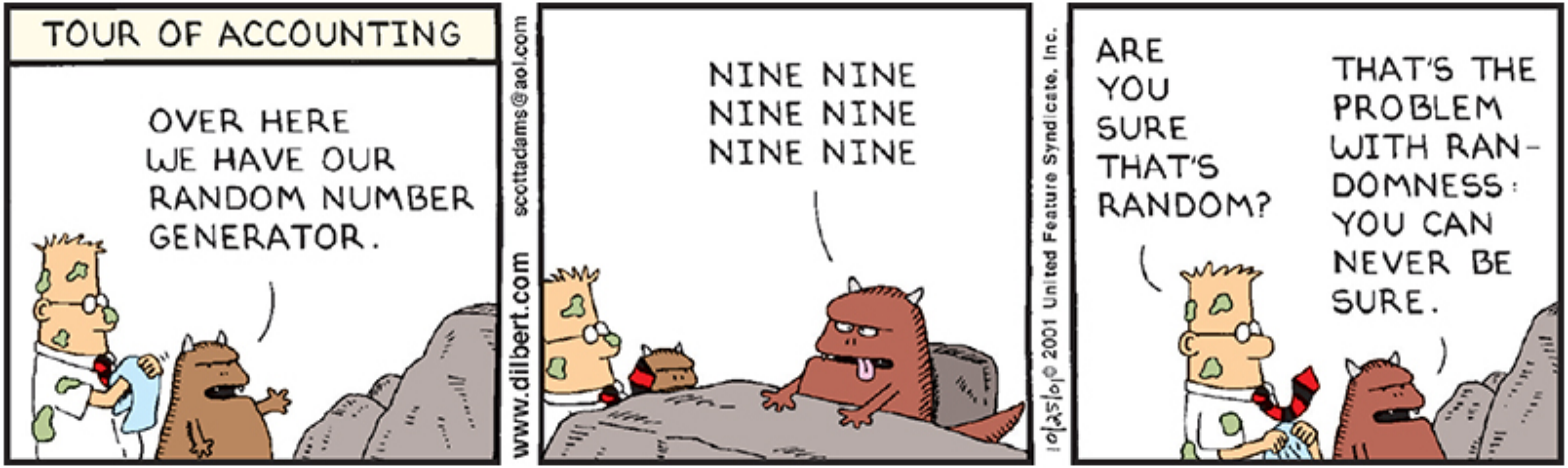
- Entropy source shared by all cores
- RS-NOR latch becomes metastable when input de-asserted
- Output settles to 0 or 1
- Feedback helps reach metastable state
- Output detects when settled, stores result, reasserts input

Intel Ivy Bridge Trustworthiness?

- Design appears to be sound
- Possible back doors?
  - Snowden leaks show NSA coerced hardware and software manufacturers to introduce back doors
  - Intel and Via hardware considered suspect
- Approach by FreeBSD
  - Use hardware PRNG as a source
  - Pass through Yarrow PRNG algorithm (3DES+SHA1) when producing cryptographically significant random numbers
- See: https://www.schneier.com/paper-yarrow.pdf



Evaluation of Random Numbers (1)



Evaluation of Random Numbers (2)

- Key requirements:
  - *Unbiased statistical distribution* 🡐 determined by statistical tests
  - *Unpredictability* 🡐 determined by modeling
  - *Unreproducibility* 🡐 hardware property

- Statistical tests for unbiased distributions:
  - Several “standard” test suites exist
  - NIST test suite
  - DIEHARD test (overlaps with NIST)
  - Individual tests such as Chi Square Test

NIST test suite

- Suite of different statistical tests, e.g.
  - Frequency tests of single bits (also relative to blocks)
  - Runs tests: check number of consecutive identical bits
  - Binary matrix rank test: check for linear dependencies of bits by constructing random matrices and computing their rank
  - Approximate Entropy Test
  - Random Excursions Test: treat random bits as random walk and check statistical properties of this random walk
  - ...

Chi Square Test (1)

- Measure how well the presumed distribution (usually uniform) is represented.
- Algorithm for the test:
  - Divide the whole interval, within which the random number would be into finite number of bins (class intervals). Assume they have same size.
  - Count the number of random numbers within each interval and calculate the “expected” number of observations
    (number of random numbers used) / (number of class intervals for uniform intervals)
  - Calculate: $X^2 = \Sigma_{i=1,\ldots,m}(\text{observed}_i - \text{expected}_i)^2 / (\text{expected}_i)$
  - The value of X^2 determines if the numbers generated represent a chosen distribution, by looking up in a table, some critical values of X^2.

Chi Square Test (2)
Example

- Assume 1000 random numbers from LCG: $x_{n+1} = 125x_n + 1 \bmod 2^{12}$
- Accept randomness if confidence $\alpha = 0.95$
- Compare computed value with $X^2_{[1-\alpha,n-1]}$ or with $X^2_{[1-\frac{\alpha}{2},n-1]}$ and $X^2_{[\frac{\alpha}{2},n-1]}$ if two-sided
- $X_0^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$ (null Hypothesis)
  - E_i: Expected number of i^{th} class
  - O_i: Observed number of i^{th} class
  - n: Number of classes, n-1: degrees of freedom
- Result: $X^2_{[0.975,9]} = 2.70 < X_0^2 = 10.38 < X^2_{[0.025,9]} = 19.023$
  ➔ Accept randomness

Class	O_i	E_i	$\frac{(O_i - E_i)^2}{E_i}$
1	100	100	0.000
2	96	100	0.160
3	98	100	0.040
4	85	100	2.250
5	105	100	0.250
6	93	100	0.490
7	97	100	0.090
8	125	100	6.250
9	107	100	0.490
10	94	100	0.360
Total	1000	1000	10.380

Chi Square Test (3)

- Most commonly used test, can be used to test for any distribution
- Errors in cells with a small e_i affect the chi-square statistic more
- Best when e_i's are equal ⇒ Use an equi-probable histogram with variable cell sizes
- Designed for discrete distributions and for large sample sizes only
  ⇒ Lower significance for finite sample sizes and continuous distributions
- If less than 5 observations, combine neighboring cells
- Two-sided test to rule out random numbers that are too perfect

Summary

- Random number are the basis for all cryptographic applications: random numbers matter!
- Attacks on many cryptographic applications are possible by attacks on PRNGs.
- There is no single reliable “independent” function to generate and evaluate random numbers.
- Combination of TRNGs and PRNGs are used in practice.
- Random sources need to be carefully evaluated.