Random Number Generation

Paolo PRINETTO

Director
CINI Cybersecurity
National Laboratory

Paolo.Prinetto@polito.it

Mob. +39 335 227529

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

Goals

- Introducing the importance of Random Numbers
- Focusing on how they are usually generated via hardware and/or software.

Prerequisites

- In order to better understand the implementation of hardware Pseudo-Random Number Generators, students should be familiar with the contents of the lecture:
  - HW_S_0.2.3 – Linear Feedback Shift Registers – LFRSs

  of the course *HW_S_0.2 – Hardware Devices* of the module *Hardware Security 0 – Background Materials*

Acknowledgments

- The presentation includes material from
  - Nicolò MAUNERO – Politecnico di Torino
  - Sam MITCHUM – Virginia Commonwealth University
  - Gianluca ROASCIO – Politecnico di Torino
  - Antonio VARRIALE – Blu5 Labs
- Their valuable contributions are here acknowledged and highly appreciated.

Outline

- Introduction
- True Random Numbers
- Pseudo-Random Numbers
- Whitening
- Random Number Validation

Outline

Random Number usage

- Random Numbers are used in a variety of applications, including, among the others:
  - *simulating* random events
  - *professional gaming*
  - end-of-production & in-field *testing* of digital systems
  - Monte Carlo *algorithms*
  - *cryptographic* applications
  - …

Cryptographic applications

- The security of *cryptographic* systems depends on some secret data that is known to authorized persons but *unknown* and *unpredictable* to others.
- To achieve the required unpredictability, ~~some~~ *randomization* is typically employed.

Cryptographic applications – Example of Random Number usage

- For generating secrets such as:
  - session keys
  - large primes for key exchange and exponentiation
  - parameters in digital signatures
  - arbitrary numbers to be used only once in a cryptographic communication (nonces)
  - …

Random Number Generators - RNGs

- The process of generating random quantities is usually referred to as *Random Number Generation*
- The involved hardware or software modules are called *Random Number Generators* (RNGs)

Random Number Generators

- A Random Number Generator (RNG) is a utility or device of some type that produces a sequence of numbers within an interval [min, max] while guaranteeing that values appear unpredictable.

[https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide]

RNG characteristics

- More technically, an RNG should have the following 3 characteristics:
  1. Each new value must be *statistically independent* of the previous value. That is, given a generated sequence of values, a particular value is not more likely to follow after it as the next value in the RNG's random sequence.

RNG characteristics

2. The overall distribution of numbers chosen from the interval is *uniformly distributed*. In other words, all numbers are equally likely, and no one is more “popular” or appears more frequently in the RNG’s output than others.

RNG characteristics

3. The sequence is *unpredictable*. An attacker cannot guess some or all of the values in a generated sequence. Predictability may take the form of *forward prediction* (future values) and *backtracking* (past values).

The importance of RNG quality in security

- The lack of quality in the RNG process can introduce severe vulnerabilities that could compromise the overall cryptographic system.
- Thus, designing a secure RNG requires a level of care at least as high as designing any other element of a cryptographic system!

Examples of attacks to RNGs

- Weak random values may result in an adversary ability to break the system, as was demonstrated, for instance, by:
  - breaking the Netscape implementation of SSL
  - predicting Java session-ids

[I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr Dobb's*, pages 66–70, January 1996]

[Z. Gutterman and D. Malkhi. Hold your sessions: An attack on Java session-id generation. In A. J. Menezes, editor, *CT-RSA*, LNCS vol. 3376, pages 44–57. Springer, February 2005]

RNG criticality

- The RNG process is particularly attractive to attackers because it is typically a single isolated hardware or software component easy to locate.
- Furthermore, such attacks require only a single access to the system that is being compromised.

Random Number Taxonomy

- Random Number are usually clustered according to their *randomness quality* as :
  - *TRN – True Random Numbers*
  - *PRN – Pseudo-Random Numbers*

Outline

- Introduction
- True Random Numbers
- Pseudo-Random Numbers
- Whitening
- Random Number Validation

- Introduction
- TRNGs Implementations
- QRNG

True Random Number Generators

- TRNGs produce a stream of truly random numbers.
- That is, knowing the generating circuit and the history of numbers generated so far is of no use whatsoever to determine the next number.
- TRNGs are thus often called *non-deterministic random number generators* since the next number to be generated cannot be determined in advance.

TRNG basic principle

- TRNGs extract randomness (*entropy*) from a physical source of some type and then use it to generate random numbers.
- The physical source is also referred to as an *entropy source* and can be selected among a wide variety of physical phenomenon naturally available, or made available, to the computing system using the TRNG.

TRNG Implementation

- TRNGs are always based on an analog property, i.e., on some unpredictable physical sources outside of any human control.
- The analog value is then often *whitened* and *scaled* to produce a uniformly distributed random number set.

TRNGs

- TRNGs are always based on an analog property, i.e., on some unpredictable physical sources outside of any human control.
- The analog value is then often *whitened* and *scaled* to produce a uniformly distributed random number set.

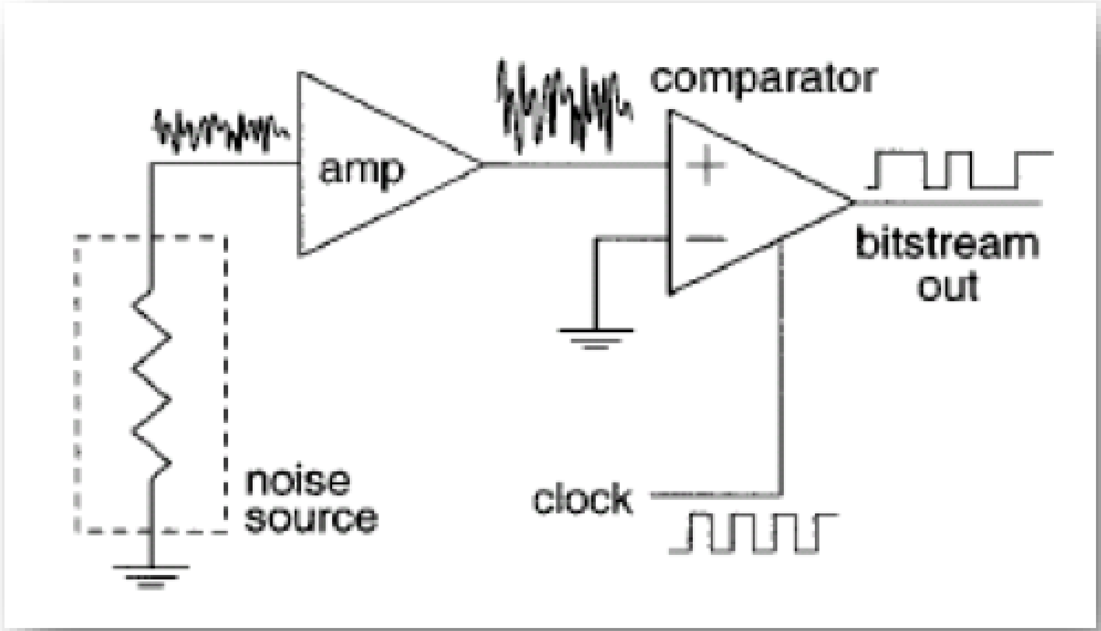
The *whitening* process will be analyzed in the sequel of the lecture, starting from slide 87

TRNG analog sources

- The most adopted solutions rely on:
  - amplifying the noise generated by a resistor (*Johnson noise*) or by a semi-conductor diode
  - feeding the noise to a comparator or Schmitt trigger
  - sampling the output of the comparator to gets a series of bits which can be used to generate random numbers.

Simple Analog RNG

- The circuitry for converting to a digital signal can be as simple as a clocked comparator



[Griffiths, Dawn, *Head First Statistics*, O'Reilly Media Inc., 2009]

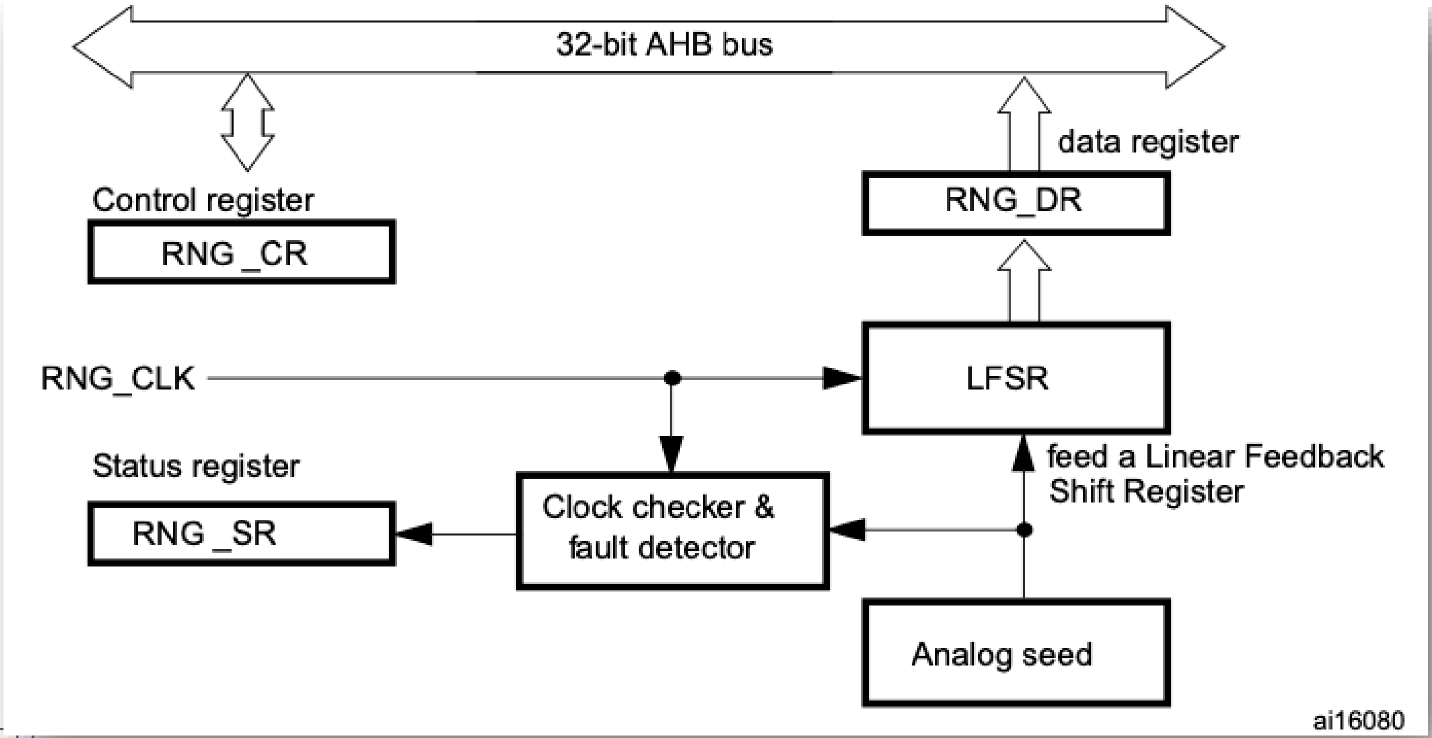
Practical implementations

- In the literature several TRNGs have been proposed, but their analysis is out of the scope of this lecture
- We shall just briefly analyze the solution adopted within the popular Cortex processors of the STM32 MCU F2 series

TRNG Implementation in the STM32 MCU F2 series

- The TRNG peripheral implemented on STM32 F2 series is based on an analog circuit.
- This circuit generates a continuous analog noise that feed an LFSR in order to produce a 32-bit random number.
- The analog circuit is made of several ring oscillators whose outputs are XORed.
- The LFSR is clocked by a dedicated clock at a constant frequency, so that the quality of the random number is independent of the MCU clock frequency.
- The contents of LFSR is transferred into the data register (RNG_DR) when a significant number of seeds have been introduced into LFSR.

TRNG Implementation in the STM32 MCU F2 series



TRNG Alternative implementations

- In some applications TRNG are implemented without resorting to external custom analog devices, but simply measuring some internal activities of the computer that are quantifiable and genuinely random.

Examples of random quantities

- Sampling the seconds value from the system clock
- Exploiting *external entropy sources* like keyboard clicks, mouse moves, network activity, camera activity, microphone activity and others, combined with the time at which they occur

Examples of random quantities

- The collection of randomness is usually performed internally by the Operating System, which provides standard API to access it (e.g., reading from the /dev/random file in Linux).

G. Zvi, B. Pinkas, T. Reinman:
”Analysis of the Linux Random Number Generator” - 2006 IEEE Symposium on Security and Privacy (S&P'06) pp. 385-403

The Linux Random Number Generator

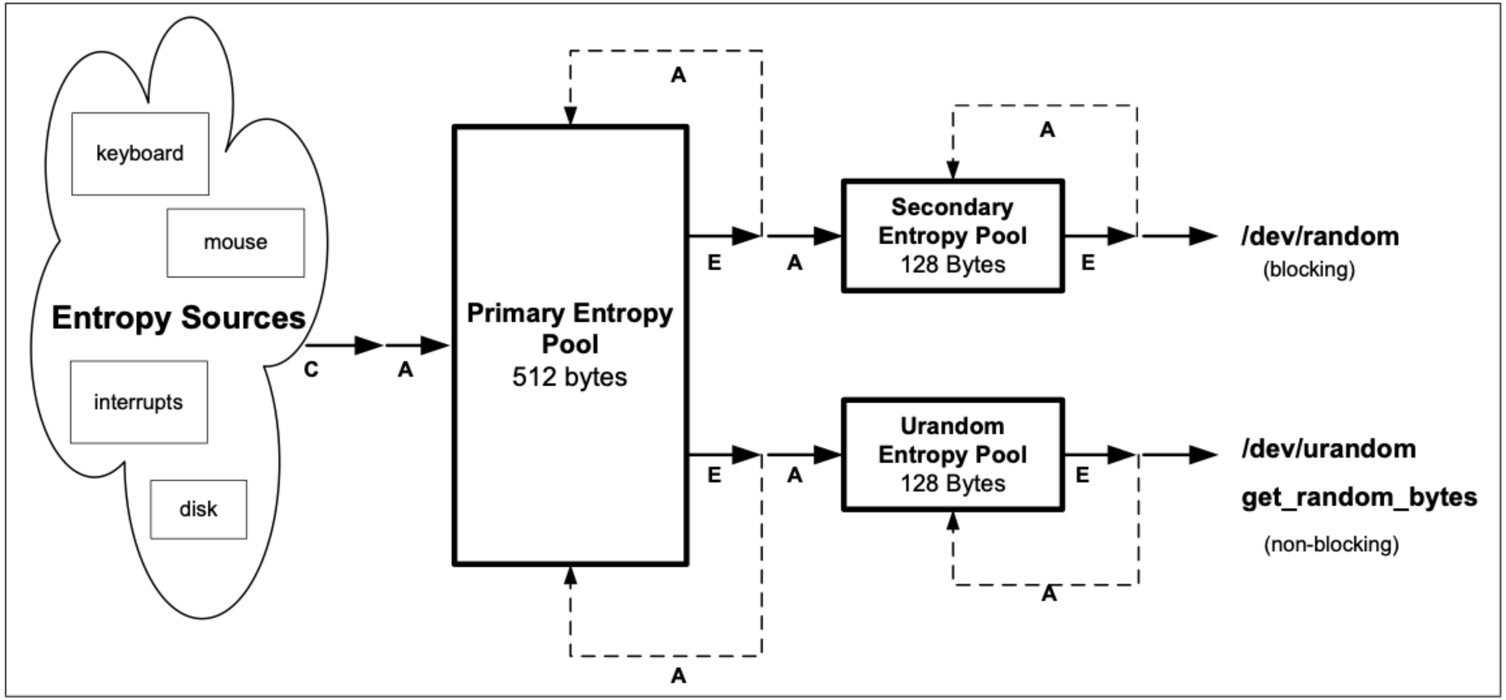


Figure 2.1: The general structure of the LRNG: Entropy is collected (C) from four sources and is added (A) to the primary pool. Entropy is extracted (E) from the secondary pool or from the urandom pool. Whenever entropy is extracted from a pool, some of it is also fed back into this pool (broken line). The secondary pool and the urandom pool draw in entropy from the primary pool.

Inefficiency sources

- Considering system clock, process scheduling, and other system effects may result in some values occurring far more frequently than others
- When dealing with the human generated entropy, values do not distribute evenly across the space of all possible values: some values are more likely to occur than others, and certain values almost never occur in practice.

Inefficiency sources (cont'd)

- The entropy in the Operating System is usually limited and waiting for more entropy is slow and unpractical.

Consequences

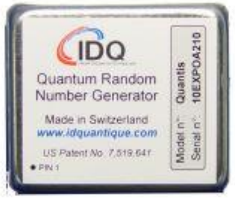
- Most cryptographic applications use *Cryptographically Secure PRNGs* (CSPRNGs), analysed later in this lecture.

Outline

Most recent TRNGs

- Recently, brand new TRNGs - the so called *QRNGs* - are made available on the market.

USB



OEM

PCIe

Quantis Appliance

Quantum RNG - QRNG

- QRNGs pull their random numbers from inherently indeterministic *quantum processes*

QRNG principle

[Xiongfeng Ma, Xiao Yuan, Zhu Cao, Bing Qi and Zhen Zhang: “Quantum random number generation” – npj Quantum information - www.nature.com/npjqi]

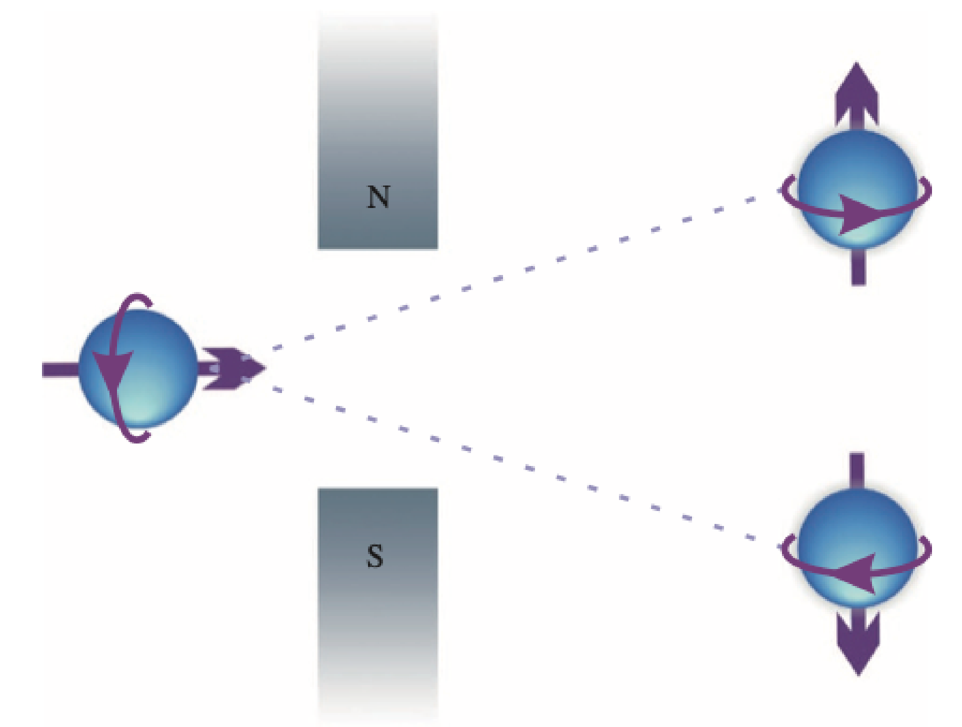
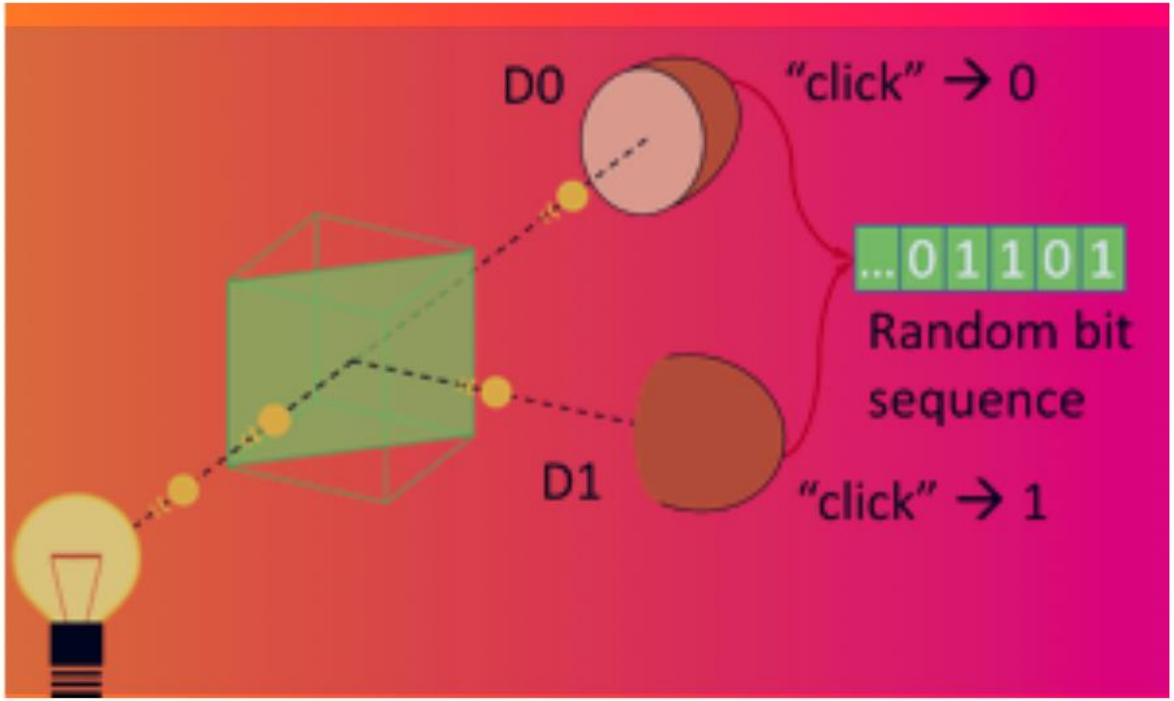


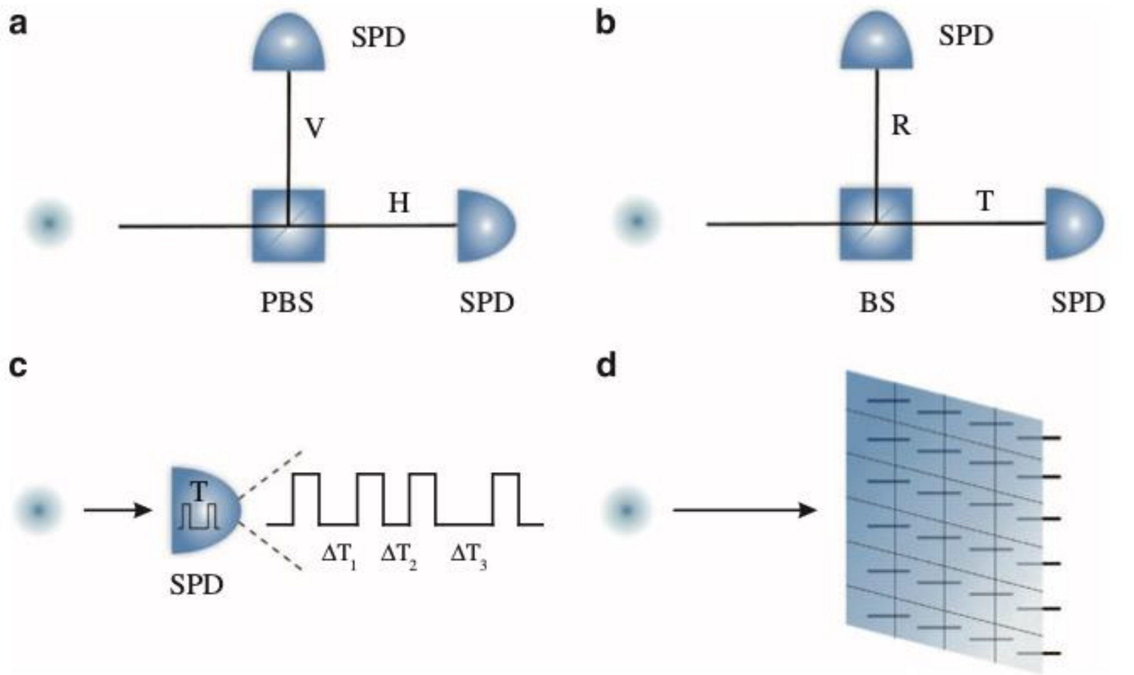
Figure 1. Electron spin detection in the Stern–Gerlach experiment. Assume that the spin takes two directions along the vertical axis, denoted by $|\uparrow\rangle$ and $|\downarrow\rangle$. If the electron is initially in a superposition of the two spin directions, $|\rightarrow\rangle = (|\uparrow\rangle + |\downarrow\rangle)/\sqrt{2}$, detecting the location of the electron would break the coherence, and the outcome ($\uparrow$ or $\downarrow$) is intrinsically random.

QRNG Implementation

- At its core, a QRNG chip contains a light-emitting diode (LED) and an image sensor.
- Due to quantum noise, the LED emits a random number of photons, which are captured and counted by the image sensor’s pixels, giving a series of raw random numbers that can be accessed directly by the user applications.

QRNG Implementation





[Xiongfeng Ma, Xiao Yuan, Zhu Cao, Bing Qi and Zhen Zhang: "Quantum random number generation" – npj Quantum information - www.nature.com/npjqi]

Figure 2. Practical QRNGs based on single-photon measurement. (**a**) A photon is originally prepared in a superposition of horizontal (*H*) and vertical (*V*) polarisations, described by $(|H\rangle + |V\rangle)/\sqrt{2}$. A polarising beam splitter (PBS) transmits the horizontal and reflects the vertical polarisation. For random bit generation, the photon is measured by two single-photon detectors (SPDs). (**b**) After passing through a symmetric beam splitter (BS), a photon exists in a superposition of transmitted (*T*) and reflected (*R*) paths, $(|R\rangle + |T\rangle)/\sqrt{2}$. A random bit can be generated by measuring the path information of the photon. (**c**) QRNG based on measurement of photon arrival time. Random bits can be generated, for example, by measuring the time interval, Δt, between two detection events. (**d**) QRNG based on measurements of photon spatial mode. The generated random number depends on spatial position of the detected photon, which can be read out by an SPD array.

Outline

Pseudo-Random Number Generator

- A PRNG is an algorithm or a hardware device that generates a sequence of random bits (or numbers), starting from an initial value, called the *seed*.

Generated sequences

- The sequence of random values generated by a PRNG:
  - *repeats periodically*
  - is typically *very long*, and it is hard to recognize that the sequence of numbers repeats cyclically
  - its periodicity depends on the *size of the internal state* model of the PRNG

Sequence length

- The best PRNG algorithms available today have such a large period that this weakness can be practically ignored.
- For example, the Mersenne Twister MT19937 PRNG with 32-bit word length has a periodicity of 2^{19937}-1

[Nishimura, Makoto, Matsumoto and Takuji: *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.* 1, January 1998, ACM Transactions on Modeling and Computer Simulation, Vol. 8]

Caveat

- Nevertheless, perfect knowledge of the generating circuit and the most recently generated numbers could enable one to guess the next number to be generated

Caveat

- Nevertheless, perfect knowledge of the generating circuit and the most recently generated numbers could enable one to guess the next number to be generated
- This is because a PRNG computes the next value on the basis of a specific internal state and a specific, well-defined, algorithm.

Consequences

- Thus, while a generated sequence of values exhibit the statistical properties of randomness (independence, uniform distribution), the overall behaviour of the PRNG is entirely predictable.
  - After a surprisingly small number of observations (e.g., 624 for Mersenne Twister MT19937), each and every subsequent value can be predicted.

Consequences

- For this reason, PRNGs are often called *Deterministic random number generators,* since, given a particular seed value, a same PRNG will always produce the exact same sequence of "random" numbers.

Seed criticality

- To ensure forward unpredictability, care must be taken in obtaining seeds.
- The values produced by a PRNG are completely predictable if both the seed and the generation algorithm are known.
- Since in many cases the generation algorithm is publicly available, the seed must be kept secret and generated from a TRNG.
- The fundamental requirement for the PNRG to work well is its seed security.

Outline

PRNG Implementations

- Several implementations are possible, resorting to:
  - *Hardware devices*
  - *Software programs*
  - *Hash Functions*

PRNG Implementations

- Several implementations are possible, resorting to:
  - *Hardware devices*
  - *Software programs*
  - *Hash Functions*

Hardware PRNG

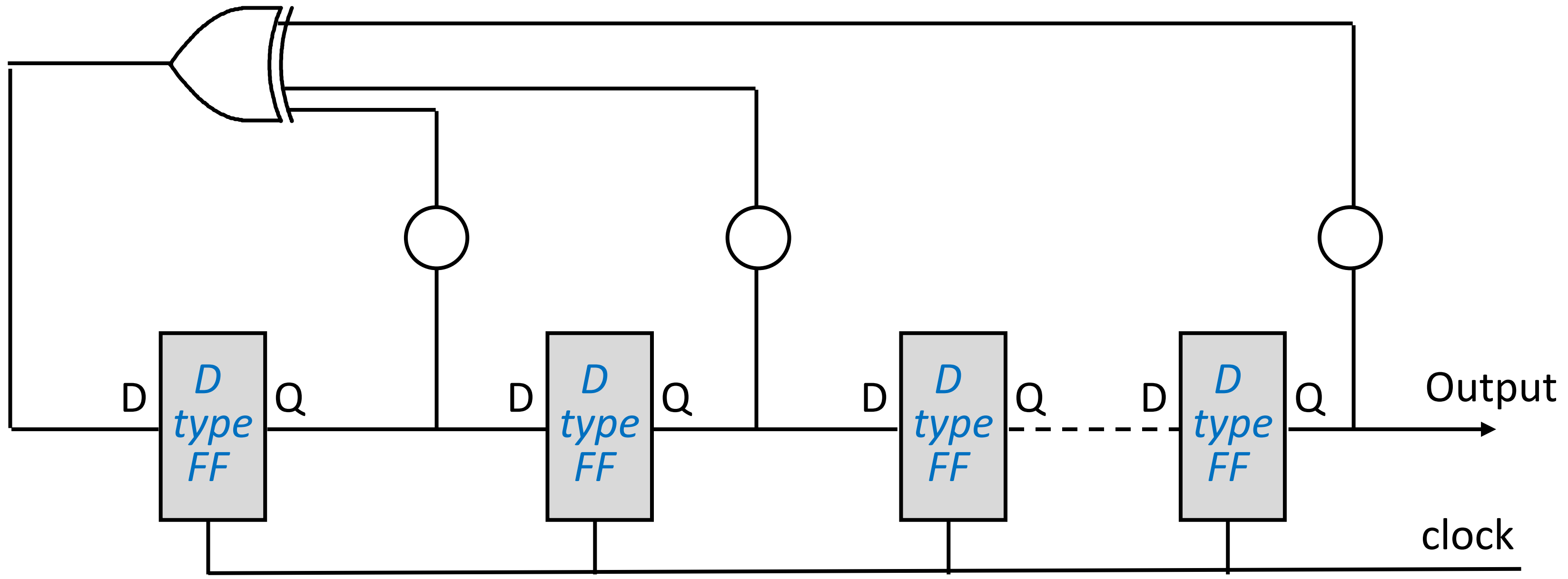
- Hardware PRNGs are mostly implemented resorting to Maximal Length *Autonomous Linear Feedback Shift Registers* (ALFSRs), i.e., an LSFR with no inputs and whose *characteristic polynomial* is primitive.

Hardware PRNG

- Hardware PRNGs are mostly implemented resorting to Maximal Length *Autonomous Linear Feedback Shift Registers* (ALFSRs), i.e., an LSFR with no inputs and whose *characteristic polynomial* is primitive.

For additional details please refer to the lecture:
HW_S_0.2.3 – Linear Feedback Shift Registers - LFRSs

Example of an n-bits ALFSR



Theorem

- From an n-bits ALFSR, by appropriately changing the characteristic polynomial (i.e., the bits to be sent to the exor port), it is possible to obtain ALL the periods between 1 and $2^n - 1$.

Maximal-length ALFSR

- A n-bits ALFSR capable of reaching all the possible 2^n - 1 states is named *Maximal-length ALFSR* and its characteristic polynomial is *primitive.*

Why are ALFRs widely used

- The output of a maximal length ALFSR:
  - appears to be random though it is actually well ordered
  - *will pass all statistical tests for randomness*.

Output sequences generated by Maximal-length ALFSR

- Starting from a state other than 00...0, all possible states are reached before having repetitions
- If a window of amplitude n is scrolled on the output m-bit sequence, each of the $2^n - 1$ possible n-uple of bits is selected once and only once
- The number of the 1's in any complete sequence differs from the number of the 0's for at most one unit.

Output sequences generated by Maximal-length ALFSR

- In each complete sequence of m bits:
  - there are as many subsequences of 1 as there are of 0
  - half of the subsequences have length 1, a quarter has length 2, an eighth has length 3, and so on
  - there are (m + 1) / 2 transitions (from 0 to 1 or vice-versa).

Practical solutions

- To mask the fact that they are deterministic, PRNGs are often devised to generate more bits than needed
- For example, a 128-bit ALFSR may be used to implement a 32-bit PRNG
- Protection comes from the fact that it is more difficult to discover the 96 hidden bits than the 32 bits used for the random number.

PRNG Implementations

- Several implementations are possible, resorting to:
  - *Hardware devices*
  - *Software programs*
  - *Hash Functions*

Software PRNG

- Software PRNGs are mostly implemented using programs that implement a recurrence operation.
- Next slide shows one of th~~os~~e most commonly used approach to generate pseudo-random integers.

Example of Software PRNG: *Linear Congruential Generators*

- The next pseudo-random integer is generated using:
  - the previous pseudo-random integer
  - integer constants
  - the module operation on integer numbers:

$$X_{n+1} = (a\,X_n + c) \bmod M$$

where:

- a, c, and M are integers
- X_n, n=1,2,..., is a sequence of integers with $0 \leqslant X_n < M$

Example of Software PRNG: *Linear Congruential Generators*

- To start, the algorithm requires an initial seed X_0
- The normalized sequence

$$un = X_n \ / \ M$$

n =1,2,..., is a random number sequence in [0, 1)

PRNG Implementations

- Several implementations are possible, resorting to:
  - *Hardware devices*
  - *Software programs*
  - *Hash Functions*

Exploiting Cryptographic Hash functions

- *Cryptographic hash functions* can be used as PRNG:

  n = hash(seed)

- The result is an effectively pseudo-random number because a cryptographic hash function takes in input a variable amount of data and outputs a fixed-length block n that, due to the avalanche effect, will contain a number of uniformly distributed bits.
- In this case, particularly "long" seeds need to be selected.

Exploiting Cryptographic Hash functions

- *Cryptographic hash functions* can be used as PRNG:

  n = hash(seed)

- The result is an pseudo-random number because a crypto data a he effect,

For additional details please refer to the lecture:
CR_2.2 - Hash functions

- In this case, particularly "long" seeds need to be selected.

Outline

Pro’s & Con’s of “Pseudo” randomness

- In some contexts, the deterministic nature of PRNGs is an advantage, since PRNGs provide a way to generate a long sequence of random data inputs that are repeatable by using the same PRNG, seeded with the same value.

Pro’s & Con’s of “Pseudo” randomness

- Examples include, among the others:
  - simulation and experimental contexts, where one can be interested in comparing the outcome of different approaches using the same sequence of input data

Pro’s & Con’s of “Pseudo” randomness

- Examples include, among the others:
  - simulation and experimental contexts, where one can be interested in comparing the outcome of different approaches using the same sequence of input data
  - digital system testing, where, at test time, the unit under test is excited with random input patterns and the corresponding values of its outputs are compared with those previously computed via simulation.

Pro’s & Con’s of “Pseudo” randomness

- In other contexts, however, the PRNG’s determinism is highly undesirable.
- Consider a server application that generates random numbers to be used as cryptographic keys in data exchanges with client applications over secure communication channels.

Pro’s & Con’s of “Pseudo” randomness

- Even with a sophisticated and unknown seeding algorithm, an attacker who knows (or can guess) the PRNG in use can deduce the state of the PRNG by observing the sequence of output values.

PRNG in cryptography

- PRNGs are considered *cryptographically insecure*.
- However, PRNG researchers have worked to solve this problem by creating what are known as *Cryptographically Secure PRNGs* (CSPRNGs).

Outline

CSPRNG requirements

- There are two major requirements for a PRNG to be a CSPRNG:
  - *Satisfy the next-bit test*: if someone knows all k bits from the start of the PRNG, he/she should be unable to predict the bit k+1 using reasonable computing resources.
  - *Withstand the state compromise extensions*: if an attacker guesses the internal state of the PRNG or it is revealed somehow, he/she should be unable to reconstruct all previous random numbers prior to the revelation.

CSPRNG Implementations

- Most CSPRNGs:
  - use a combination of entropy from the OS and high-quality PRNG generator
  - are often “reseed”: when new entropy comes from the OS (e.g., from user input, system interruptions, disk I/O or hardware random generators), the underlying PRNG changes its internal state based on the new entropy bits coming.
- This constant reseeding over the time makes the CSPRNG really hard to predict and analyse.

CSPRNG Implementations

- Several CSPRNG algorithms have been proposed, based on:
  - applying a cryptographic hash to a sequence of consecutive integers
  - using a block cipher to encrypt a sequence of consecutive integers ("counter mode")
  - XORing a stream of PRNG-generated numbers with plaintext ("stream cipher")

CSPRNG Implementations (cont'd)

- number theory, relying on the difficulty of the integer factorization problem (IFP), the discrete logarithm problem (DLP) or the elliptic-curve discrete logarithm problem (ECDLP)
- specific algorithms , such as:
  - *Yarrow* algorithm is incorporated in iOS and macOS for their /dev/random devices
  - *Fortuna* algorithm, published in 2003 by Bruce Schneier and Niels Ferguson and used in the FreeBSD OS

CSPRNG in Intel® processors

- Intel® 64 and IA-32 processors implement a CSPRNG resorting to a mix of hardware and software components

[https://software.intel.com/en-us/articles/the-drng-library-and-manual]

[https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide]

Intel® Digital Random Number Generator (DRNG)

- *Intel® Secure Key*, code-named *Bull Mountain Technology*, is the Intel name for the Intel® 64 and IA-32 Architectures instructions RDRAND and RDSEED and the underlying Digital Random Number Generator (DRNG) hardware implementation.

CSPRNG in programming

- Always use cryptographically secure random generator libraries, like:
  - the *java.security.SecureRandom* in Java 8
  - the *secrets* library in Python

[https://cryptobook.nakov.com/secure-random-generators]

Outline

Linear combination of variances

- According to statistics:
  - the variance of the linear combination ^ of two sets of numbers is equal to the sum of the variances of the individual sets as long as there is no correlation between the two sets:

$$V(x \wedge y) = V(x) + V(y)$$

  - The linear combination can be realized by adding, subtracting, or XORing the two sets.

[Griffiths, Dawn, *Head First Statistics*, O'Reilly Media Inc., 2009, pp. 230]

Variance

- Variance (σ^2) in statistics is a measurement of the spread between numbers in a data set
- That is, it measures how far each number in the set is from the mean and therefore from every other number in the set.

Practical implications

- Since a TRNG is independent of any LFSR based PRNG by virtue of its definition, the TRNG could be linearly combined with the PRNG and the resulting combination would be more statistically random than either of the two input streams.

Whitening

- The process of combining the output of the TRNG with the output of a LFSR in order to increase the variance, and hence the randomness of the TRNG.

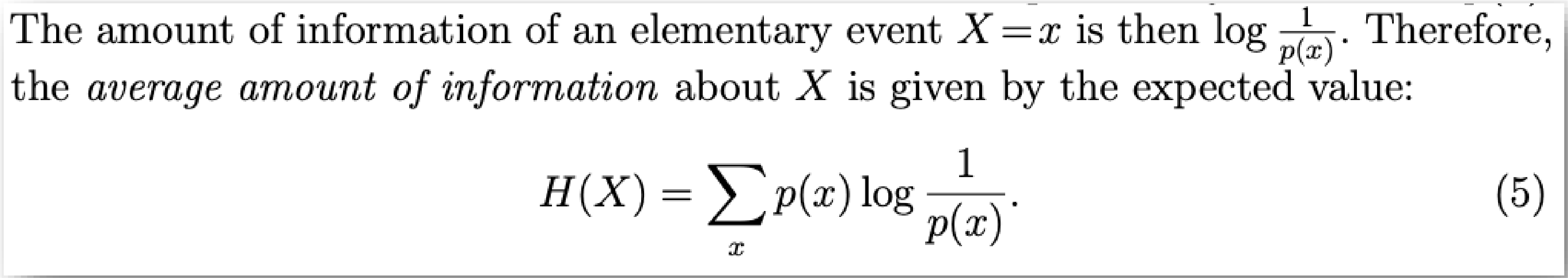
Outline

Tests for Randomness

- There are many ways to test the randomness of a stream of numbers.

From a *mathematical* point of view…

- The quality of randomness can be measured in bits by ~~the~~ means of *Shannon entropy H(X)* of a random variable X with probability distribution p(x)



The amount of information of an elementary event $X = x$ is then $\log \frac{1}{p(x)}$. Therefore, the *average amount of information* about X is given by the expected value:

$$H(X) = \sum_x p(x) \log \frac{1}{p(x)}. \tag{5}$$

[Olivier Rioul: This is IT: A Primer on Shannon's Entropy and Information – L'Information, Seminaire Poincaré XXIII (2018) 43

From a *practical* one…

- A few simple ways to test the randomness of a stream of numbers would be:
  - count the number of ‘1’ bits and the number ‘0’ bits and make sure they are approximately the same
  - break the stream into groups of say four bits and make sure that each possible 4-tuple occurs roughly the same number of times (0000, 0001, 0010, 0011, ... 1111)
  - pick a bit size and a particular pattern for that size and count how many bits are produced before that exact pattern is produced again.

Tests for Randomness

- Several tests have been developed and standardized.
- They are usually clustered into 2 classes:
  - *Validation Tests*
  - *Diagnostic Tests*

Validation vs Diagnostic Tests

- *Validation Tests* are applied to big sequence of bits to measure the quality of a random noise generator
- *Diagnostic Tests* are applied to short sequences of bits (at least 20.000 bits) to quickly detect whether the noise is compromised. Although they identify “bad” noise, they do not guarantee any appropriate noise significance level.

Validation Tests

- To guarantee that the output of an RNG is really random, several test suites have been created to provide requirements and references for their construction, validation, and usage.
- The most common test suites are:
  - Diehard
  - “ent”
  - NIST Statistical Test Suite (Special Publication 800-22 Revision 1a)

Diehard Suite

- The Diehard Suite is a set of tests developed by George Marsaglia and published in 1995
- The suite is made up of several statistical tests:
  - Birthday Spacing, Overlapping Permutations, Ranks of Matrices, Monkey Test, Count the 1s, Parking Lot Test, Minimum Distance Test, Random Sphere Test, The Squeeze Test, Overlapping Sums Test, Runs Test, The Craps Test

The “ent” suite

- The ”ent” suite is an utility developed by Fourmilab which applies several tests to sequence of bytes stored in a file

- The suite includes the following calculations and tests:
  - Entropy
  - Chi-square test
  - Arithmetic Mean
  - Monte Carlo Value for Pi
  - Serial Correlation Coefficient.

The NIST SP800-22b statistical test suite

- The NIST SP800-22b statistical test suite “sts-2.1.1” is a software package developed by the US National Institute of Standards and Technology (NIST) to probe the quality of random generators for cryptographic applications.

[“A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications” - https://www.nist.gov/publications/statistical-test-suite-random-and-pseudorandom-number-generators-cryptographic]

[http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html]

The NIST SP800-22b statistical test suite

- It consists of 16 tests developed to test the randomness of a binary sequence (they are summarized for references in the next 3 slides).
- The scoring is somewhat arbitrarily; for instance, NIST suggests a threshold of 96%; that is, a score of 95.9% fails while a score of 96.0% passes.

The NIST SP800-22b statistical test set

1) **Frequency Test**: This test compares the proportion of 1’s to 0’s in the data. The proportion of 1’s should be about half the number of bits. The test fails if there are too many or too few 1’s in the bit stream.

2) **Block Frequency Test**: This test computes the proportion of 1’s to 0’s in a specified block size. For random data the frequency should be about half the block size. This test fails if there are too many blocks which have either too many or too few 1’s.

3) **Cumulative Sums Test**: This test identifies the maximal excursion from 0 of a random walk using the values [-1, +1]. In other words, start at a point in the bit stream and move forward to the adjacent bit. If it is a “0” then SUM = SUM – 1. If it is a “1” then SUM = SUM + 1. If the bits alternated perfectly then the cumulative sum would remain low. If there are too many 1’s or 0’s in a row, however, the cumulative sum gets large. This test fails if the cumulative sum is either too large or too small.

4) **Runs Test**: This test counts the number of occurrences of runs of 1’s. A run is defined as a continuous stream of bits of the same value bounded at the start and the end by bits of the opposite value. The expected results are more runs of shorter numbers of 1’s and fewer runs of longer numbers of 1’s. The test fails if there is significant deviation from the expected number of runs for any length of consecutive bits.

5) **Longest Runs Test**: This test counts the longest number of consecutive bits in each block of m bits. The test fails if there are too many consecutive 1’s in the block.

The NIST SP800-22b statistical test set

6) **Rank Test**: This test divides the stream of binary bits into rows and columns of matrices. It then calculates the rank of each resulting matrix as a way of testing for linear dependence – hence too many repeated patterns. The test fails if ranks of the resulting matrices are incorrectly distributed.

7) **Discrete Fourier Transform Test**: This test examines the peak heights in the discrete Fourier transform of the sequence. The purpose is to detect repetitive patterns in the sequence. The test fails if the number of peaks exceeding a given threshold is too large.

8) **Non-overlapping Template Matching Test**: This test searches the bit stream for specific, aperiodic patterns. If the pattern is found, the search is started again just beyond the end of the pattern. If the pattern is not found, the search is started again at the next bit position. The test fails if too many occurrences of the pattern are found.

9) **Overlapping Template Test**: This test is similar to the non-overlapping template matching test except if the pattern is found, the search is continued from the next bit following the start of the pattern so that patterns which overlap are detected. Again the test fails if too many occurrences of the pattern are found.

10) **Maurer's Universal Statistical Test**: This test counts the number of bits between matching patterns in the data stream. This measure is related to how well the stream can be compressed. The test fails if the bit stream is compressible.

The NIST SP800-22b statistical test set

11) **Approximate Entropy Test**: This test compares the frequency of occurrence of all patterns of a certain bit length with the frequency of occurrence of all patterns that are one bit longer. The test fails if the difference in frequency of occurrence for the two lengths is not as expected for random data.

12) **Random Excursions Test**: This test is similar to the cumulative sum test in that a sum is calculated by taking a random walk from a point considered to be the origin and returning to that point. For each bit traversed, subtract 1 if the bit is a “0” and add 1 if the bit is a “1”. The test actually examines eight different measurements – how many times each of the sums in the set [-4, -3, -2, -1, +1, +2, +3, +4] are encountered during a random walk. The test fails if the number of times each sum is encountered does not match that predicted for random data.

13) **Random Excursions Variant Test**: This test is a more stringent variation of the random excursions test. The difference is the number of sums. This test uses a total of eighteen sums, [-9, ... -1, +1, ... +9] where the random excursions test only uses eight. The test fails when the number of times each sum occurs does not match that expected for random data.

14) **Serial Test**: This test measures the frequency of occurrence of all possible overlapping patterns of a specified bit size. In a random stream, each pattern should occur approximately the same number of times. The test fails if the number of occurrences of each pattern is not approximately the same. Note for the case of 1-bit patterns, this test degenerates to the frequency test.

15) **Lempel-Ziv Test**: This test counts the number of cumulatively distinct patterns in the sequence. It is a measure of how much the bit stream can be compressed. The test fails if the bit stream can be compressed.

16) **Linear Complexity Test**: This test calculates the size of a LFSR that would be required to produce the bit stream. The test fails if the required LFSR is too small.

Diagnostic Tests

- The most used diagnostic test suite is described in the FIPS 140-2 standard:
  - Easy to be implemented in embedded systems
  - Fast detection (suitable for both boot-level and run-time diagnostic)

FIPS 140-2 diagnostic tests

- The FIPS 140-2 diagnostic tests include:
  - *Monobit Test*: the number n of 1's in the bitstream must be 9,725 < n < 10,275
  - *Poker Test*: determines whether the sequences of length n (n=4) show approximately the same number of times in the bitstream
  - *Runs Test*: determines whether the number of 0's (gap) and 1's (block) of various lengths in the bitstream are as expected for a random sequence
  - *Long Run Test*: is passed if there are no runs longer than 26 bits

Diagnostic Tests Usage

- Thanks to their simplicity, they are typically performed “in-the-field” in order to detect attacks to the RNG, by checking the quality of the generated sequences.

Paolo PRINETTO

Director
CINI Cybersecurity
National Laboratory

Paolo.Prinetto@polito.it

Mob. +39 335 227529

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it