Strong(er) Randomness

Elisabeth Hanning, Axel Pyk
Email: {eliha589,axepy855}@student.liu.se
Supervisor: Jan-Åke Larsson, {jan-ake.larsson@liu.se}
Project Report for Information Security Course
Linköpings university, Sweden

1 Abstract

John von Neumann, one of the first mathematicians to extensively study random sequences, once said: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."[1]

This report addresses the subject of randomness and random numbers, how they are created and used in different areas of computer science. Since randomness is a vital part of security this report focus in, how random numbers used in cryptography are algorithmically created, what kind of threats exist against pseudo random number generators and how it is possible to verify randomness.

The three main problems discussed are:

1. *How are sequences of random numbers algorithmically created?*
2. *Can a random number generator be verified?*
3. *What are the common attacks on pseudo random number generators?*

The method of work has been a literature study of relevant articles in the area of randomness and pseudo-random number generation.

2 Introduction

Random numbers are today used in many different areas of computer science and especially in the area of information security. It is for example used for creating session keys, random IDs and white noise. Another area within information security where random numbers are vital is cryptography, example for creating keys. Since there are a lot of security mechanisms depending on random numbers it is important that they are truly random. A problem occurs if the system depends on a weak random number generator, one that an attacker can manipulate or figure out the subsequent number.

The report gives first an introduction about what a random number is and what is can be used for. Then follows a more detailed analysis of how random numbers are created using software and external entropy sources. Regarding using software to create random numbers a deeper study of the Yarrow-160 design principles, incorporated in many unix-like operating systems, including the Linux kernel is performed, with an additional security analysis of the implementation of the Linux Pseudo-Random Number Generator (LRNG).

The second part of the report is about verifying a random number. The certifications that exist regarding random numbers and the common criteria and standard test that are accepted is discussed.

In the third part of the report common attacks on pseudo-random number generators are discussed. These attacks can be divided in three different types; direct cryptanalytic attack, input based attack and state compromise extension attacks.

3 Background

Generating random numbers can be considered a trivial task for humans as it has been performed since ancient times. By rolling a fair dice or flipping a fair coin it is impossible to predict the outcome, meaning the result is truly random. But these techniques are not efficient in combination with machines, and when computerizing the creation it becomes much more complicated. The problem arises from the fact the computer operates by generating output data from an input data, and as long the computer is in the same state, the outcome will be identical. This means that the computer is a deterministic environment. But it is hard to find non-deterministic environments in the nature due to the fact laws rule the nature. True randomized behaviour can first be found in quantum mechanics.

Deterministic randomness, also called pseudo randomness are sequences generated by arithmetical methods that appears to be random, thereof the 'pseudo'-prefix. The programs generating the number are called pseudo random number generators (PRNGs). They trace back to the late 1940's when John von Neumann reinvented[1] the middle-square method, as a complement to his work on the hydrogen bomb. PRNGs suffers from two downsides: first, they are limited to a finite number generated numbers, meaning they are periodic and second, they are reviewable.

John von Neumann's middle-square method, traces back to the 13th century[2] and the algorithm is very simple. Initially a seed is chosen $s_0 = mid(\sigma, m)$, where σ for instance can be seconds since noon, with a desired length of the outcome m digits. Next generated number is given by $s_{n+1} = mid(s_n^2, m)$, with condition; if $len(s_n) < m$, leading zeroes is added to s_n. The algorithm's period roots from number of digits in the desired output m and limits the outcome to 2^m generated numbers.

To obtain true randomness within a computer it is possible to use a so called, True Random Number Generator (TRNG). TRNGs generates random numbers from white noise by physical processes, such as thermal noise, the photoelectric effect and other quantum phenomena.

"Although randomness can be precisely defined and can even be measured, a given number cannot be proved to be random. This enigma establishes a limit to what is possible in mathematics." - Gregory J. Chaitin[3].

Consider two numbers:

$$\begin{cases} X_1 = 1010101010 \\ X_2 = 0110110011 \end{cases}$$

where X_1 is '10' duplicated five times and X_2 is generated by tossing a coin ten times. If we would ask a person what they think next digit in X_1 would be, most would say '1'. Does that mean X_2 is random and X_1 is not? No. The provenance of the series does not certify that it is random. Both values occurs with a probability of 2^{-10}, meaning it should not be more surprising to acquire series with patterns than series without patterns. The conclusion contradicts the possibility of distinguishing the random from the orderly and more paradigms are necessary to distinguish randomness.

There are three main approaches to achieve algorithmic randomness according to Fiala, Kratochvil & Koubek [9]:

1. The incompressibility paradigm,
2. The measure-theoretic paradigm, and
3. The unpredictability paradigm.

3.1 The incompressibility paradigm

1965 introduced A. N. Kolmogorov a property of all infinite random sequences, *algorithmically random sequences*, called compressibility/complexity[4].

Kolmogorov compressibility. *Given a binary sequence $s \in \{0,1\}^n$, define the plain Kolmogorov compressibility of s as $C(s) = min\{|\tau| : U(\tau) = s\}$. The basic facts concerning C are that : (i) the choice of U matters only an additive constant in the theory and (ii) for all $s, C(s) <= |s| + O(1)$.*

Which indicates Kolmogorov defined the randomness as the entropy H of a sequence with length K, where the randomness of s increases as the entropy $H(s) \to K$.

This tells us that iif the sequence s not can be represented with a sequence smaller then the length of s, $K(s) >= |s|$, it is so called Kolmogorov random or *incompressible*.

X_1 can be represented as "10" $*5$, which is represented by a string with length 6 and is therefore not Kolmogorov incompressible. X_2 can not be represented in another way than 0110110011 and is therefore Kolmogorov incompressible.

The problem with Kolmogorov incompressiveness is, it is incomputable.

3.1.1 Entropy and randomness

As mentioned in 3.1, indicated Kolmogorov randomness as the entropy of a sequence. This means it is possible to get an indication about the randomness of a word.

If a 32-bit word is completely random, its entropy is 32 bits. But if the word only takes four different values, each with a possibility of 25%, the entropy is 2 bits. This means the entropy does not measure how many bits there are in a word, it is a measure of the **uncertainty** of the value.

If an attacker finds out that the 32-bit word contains 18 zeros and 14 ones, there are about $2^{28.8}$ possible values, compared to 2^{32} earlier. This limits the entropy to maximum 28.8 bits.

3.2 The measure-theoretic paradigm

1966 published the Swedish mathematician Per Martin-Löf an article called *"The definition of Random Sequences"* where he defined random sequences as followed[9]. His definition is represented by the measure-theoretical paradigm.

Martin-Löf randomness. *A given sequence $s \in \{0,1\}^n$ is Martin-Löf null if there is a uniformly c.e. sequence $\{U_i\}_{i\in n}$ such that $\mu(U_n) < 2^{-n}$.*
Given iff $s \in 2^n$ is not Martin-Löf null, $s \in 2^n$ is Martin-Löf random.

This indicates Martin-Löf defined a random sequence as a sequence with no "effectively rare properties". By effectively rare properties he states a random sequence shall not have any property that is uncommon, for instance, the distribution should be uniform between zeroes and ones in an infinite sequence.

3.3 The unpredictability paradigm

Probably the most common saying when talking about randomness, is to characterize a sequence as random if it is impossible to predict the next outcome. One way to define this is to use a martingale, an class of betting strategies.

Definition. *A martingale is a function $d : 2^{<n} \to \Re^+$ that satisfies for every $s \in 2^{<n}$ the averaging condition.*

$$2d(s) = d(s_0) + d(s_1)$$

By denoting the martingale as a betting strategy, the function d is assigning a proportion of the capital to bet on the string s.

These three paradigms defines randomness and makes it is possible to create and test randomness. By applying them when designing the programs generating random numbers within a deterministic environment, it should be possible to create sequences of pseudo random numbers.

4 Pseudo Random Number Generators

A PRNG is a program in a deterministic environment with the task of creating sequences of pseudo random numbers. L'Ecuyer defines a Pseudo Random Number Generator as following[6]:

Definition. *A (pseudo)random number generator is a structure $\Gamma = (S, s_0, T, U, G)$, where S is a finite set of states, $s_0 \in S$ is the initial state, the mapping $T : S \to S$ is the transition function, U is a finite set of output symbols, and $G : S \to U$ is the output function.*

A PRNG's state s_n always originates from s_0, also known as the seed, by recurrence $s_n = T(s_{n-1}), n \in \mathbb{N}$. The output in step n is given by $u_n = \Gamma(s_n)$. Since S is finite, the sequence s_n is periodic, which leads to $s_{p+n} = s_n$, and $u_{p+n} = u_n$ where p is the period length.

Properties required of pseudo random number generators. PRNGs must be secure against internal and external attacks. When designed, implemented and used properly, an attacker with complete knowledge over the algorithm and massive computer power still should not be able to predict the state of the PRNG. Below is the most basic requirements listed, using common terminology.

- *Pseudo randomness.* The generator's output looks random to an outside observer.
- *Forward security.* If an attacker who learns the internal state of the generator at a specific time, (s)he does not learn anything about previous outputs.
- *Break-in recovery / backward security.* If an attacker learns the internal state of the generator at a specific time, (s)he does not learn anything about future outputs of the generator, provided that sufficient entropy is used to refresh the generator's state.

There are different types of PRNGs with different intentions. This report will focus on the Yarrow-160 design, widely used in unix-based operating systems to achieve cryptographic secure PRNGs and an analysis of the implementation of Yarrow-160 in the Linux kernel, in this report called LRNG. To get a perspective a smaller example of a PRNG not designed to be used in cryptography is provided and why it is weak.

First an example of an insecure type of PRNGs and why it is insecure.

4.1 Linear Congruential Generator

By using linear equations when generating random sequences it is possible to create a PRNG with the advantages of being fast and require minimal memory to retain state, often not more than 32 or 64 bits. The programs generating the sequences are called Linear Congruential Generators (LCGs). The downside are they are weak and not suitable for cryptography due to the serial correlation. The transaction function T in LCGs is defined as followed:

$$s_n = (as_{n-1} + c) mod\ m \quad m, a, c \in \mathbb{N}$$

Below is an example to give an brief insight in how Java's standard PRNG works and why it is weak.

4.1.1 java.util.Random

Java's standard PRNG is found in the Random-class, located in Java's utilities package, called java.util[7]. The PRNG operates with the precision of 48 bits, given by m in the source code, of the seed with static constants in the transaction function T. To ensure the user never can view the PRNG's state s_n, it has a limit of 32 bits integers as maximum outcome. This means the 48 bit seed must be converted to a 32 bit integer, which is solved by bit shifting the seed to the right 16 bits. By bit shifting the seed, one generated integer is not enough to determine the seed. But if an attacker would get hold of two generated numbers, (s)he easily can brute-force the seed, and eventually be able to regenerate all previous generated keys, as well as all future numbers. Since this severe problem violates the forward security requirement and the backward security requirement, it is obvious java.util.Random is not designed to be used within cryptography. Here one of PRNGs downside becomes obvious, the code are completely reviewable and since the PRNG use a linear equation reverse engineering is possible, and fairly easy.

4.2 Reseeding techniques

It is obvious, LCGs are not supposed to be used in cryptography, rather for creating white noise. But there are design techniques to make PRNGs more secure. One approach is to collect information from non-deterministic sources and use the gathered information to regularly reseed the PRNG, making it harder for an attacker to distinguish the internal state. The information can be gathered by monitor interruptions of the mouse and keyboard, disc- and network activities and other non-deterministic sources.

There are different design principles to accomplish safer random number generators using reseeding, but this report discuss the Yarrow-160, a common design used in many systems, and its successor; Fortuna. Both rely on external non-deterministic sources for the entropy accumulation.

4.2.1 Design philosophy

When creating PRNGs using entropy accumulation to reseed the internal state, there are two basic design approaches:

The **first approach** assumes the possibility of accumulating and process enough entropy from

the samples to provide at least one bit of real entropy per output. If more output is required than entropy collected the PRNG stops and waits until the mechanism has regain stability. This means in this design, the entropy is accumulated to be immediately reused as output and the PRNG can be seen as a buffer, containing entropy accumulated from different sources. The strength of this approach is when implemented properly and a sufficient stream of entropies, there is a possibility of providing unconditional security.

The **second approach** is to accumulate enough entropy to initially put the PRNG in a unpredictable state, and the purpose of continue accumulating is to recover if the key is compromised. The strength of this approach is the performance gained compared to the first approach, to the price of security.

5 Yarrow-160 design

The Yarrow-160 design was published 1999 by Bruce Schneier, John Kelsey and Niels Fergusson[15]. The result of their report became design principles to PRNGs, that are easily implemented and better resisting existing attacks and the design is widely used in PRNGs today. The algorithm was developed using an attack-oriented process with attacks in mind from the beginning.

The goal of Yarrow-160 was not to increase the number of security primitives rather leverage the existing ones. To achieve this Yarrow-160 rely on one-way hash functions and block ciphers as much as possible.

5.1 Components

The Yarrow-160 design is divided into four independent components to fit as many systems as possible.

The components are defined as followed:

1. An **Entropy Accumulator** collecting and storing samples from entropy sources in two pools.
2. A **Reseed Mechanism** which periodically reseeds the key with new entropy from the pools.
3. A **Generating Mechanism** generating PRNG outputs from the key.
4. A **Reseed Control** determining when a reseed is to be performed.

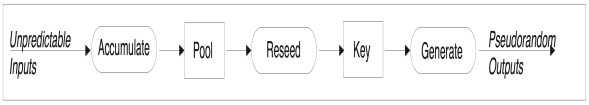

Figure 1: Generic block diagram of Yarrow-160[15]

The **Entropy Accumulator** collects entropy sources into two pools; a *fast pool* which reseeds the key frequently to reduce the duration of possible key compromises and a *slow pool* which provides rare, but conservative reseeds of the key to ensure the security of the reseed even when entropy estimates are optimistic.

Both pools of Yarrow-160 uses the cryptographic hash function **SHA-1**, designed by United States National Security Agency (NSA) with the output limit of 160 bits, thereof Yarrow-**160**. Because of the output limit of 160 bits, naturally no more than 160 bits of entropy can be collected in the pools. This determines the strength of the design.

To be able to distinguish when to reseed in the Reseed Control, Yarrow-160 performs an entropy estimation to determine how much work it would take an attacker to guess the current content of the pools. This is performed by estimating each entropy collected by the accumulator and summarizing the result.

The **Reseed Mechanism** performs the reseeding process of the pools.

When the fast pool is to be reseed it uses the current key and the hash of all inputs to the pool since the last reseed when generating a new key.

When the slow pool is to be reseed it uses the current key, the hash of all inputs to the fast pool and the hash of all inputs to the slow pool when generating a new key.

After a reseed is performed the pool is restored and cleared of all information.

The **Generating Mechanism** generates the next output by turning the block cipher into a stream cipher and encrypting the key with a counter using a symmetric-key block cipher **Triple Data Encryption Algorithm (TDEA)** *(also called Triple DES and 3DES)*, displayed below.

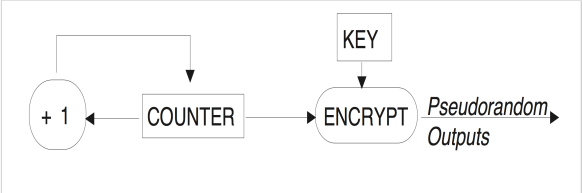

Figure 2: Generating Mechanism in Yarrow-160[15]

The **Reseed Control** determines when to reseed a pool, either when an client asks for a reseed or when the entropy estimation considers it is necessary. The process is enabled when the estimation entropy is over a threshold value; 100 bits of one source for the fast pool and 160 bits of two sources for the slow pool.

5.2 Variants of Yarrow-160

Different variants of Yarrow-160 has been devised with different hash functions (SHA-2, Davies–Meyer & Blowfish/AES/DES) and block ciphers (Blowfish, AES, 3DES) to strengthen the design or improve the performance.

Many operating systems incorporate variants of the Yarrow-160 design in their standard PRNG, and among them is the Linux kernel. To distinguish the security of the Yarrow-160 design, it is necessary to analyse an existing implementation. This report analyse the Linux Pseudo-Random Number Generator, in this report called **LRNG**.

5.3 Security analysis of LRNG

The Linux kernel is an open source project developed in the last 15 years, by a group led by Linus Torvalds and the kernel is the common element in various Linux distributions, on all types of devices. The kernel has two devices handling the creation of random numbers: **/dev/random** and **/dev/urandom**, based on the Yarrow-160 design.

/dev/random outputs more secure random numbers, and is based on the first design philosophy, mentioned in 4.2.1. This means the LRNG block the user until it has regain stability, if not enough entropy is accumulated.

/dev/urandom outputs less secure random numbers and uses the second design philosophy, mentioned in 4.2.1, making the PRNG never to block an user.

Structure. Since the LRNG has two interfaces, it has an additional pool. The structure is as followed: A primary pool (size 512 bytes), provides entropy to two smaller pools when they do not have enough entropy:

A secondary pool (size 128 bytes), provides entropy to the secure device /dev/random.

A urandom pool (size 128 bytes), provides entropy to the less secure device /dev/urandom.

Accumulated entropy is primarily added into the primary pool, if full into the secondary pool, but never the urandom pool. Whenever entropy is extracted from a pool, it is fed back into the pool again.

Because the about 2500 lines of code are viewable, its security assumable should be easily analysed. But the code is not well documented and there is no clear description of the implemented algorithm.

Zvi Gutterman, Benny Pinkas and Tzachy Reinman published 2006 a security analysis based on the LRNG, version 2.6.10 of the Linux kernel, which was released on December 24, 2004[27]. The analysis focused on four aspects:

1. A cryptanalytic attack on the forward security of the LRNG
2. An analysis of the entropy added by system events
3. Observations on the insecurity of the LRNG in the OpenWRT Linux distribution for routers
4. Observations on security engineering aspects of the LRNG, including a denial-of-service attack

5.3.1 Forward Security

The LRNG computes an output from a pool ***after*** the state of the pool is updated. Given this, it is possible to compute the output extracted from the pool at time $t-1$, if the state of the pool at time t is known. This is a violation against the forward requirement, since an attacker who knows

the internal state of the pool at time t, can compute the last output of the LRNG until the last time the pool received an entropy update.

5.3.2 Entropy Measurements

The analysis found a severe bottleneck in LRNG, based on the entropy estimation performed in Yarrow-160. On a Linux machine only accumulating entropy from the disc activity, there was an average delay of about 15 minutes between pairs of events which had a positive contribution of the entropy count, with an average size of 16 bits. Due to the low phase of positive contribution of the entropy count, this severely affects the blocking, more secure interface /dev/random.

5.3.3 Analysis of the OpenWRT Linux Distribution

Operating systems in routers are limited by their number of entropy sources, since they do not have a mouse, keyboard or hard drive (flash cards cannot be used for entropy accumulation) and the LRNG accumulate entropy from the network devices. The state of the LRNG in the OpenWRT Linux distribution resets on every reboot to a predictable value (the time of day and a constant), making the LRNG very weak.

5.3.4 Security Engineering

Denial of service. Since there is no limit on the number of bits an user can read from the devices per time unit, denial of service attacks, both internal and external, are a severe security problem against the LRNG. Below is two examples on denial of service attacks against the LRNG, one internal and one external.

By requesting the secure interface, /dev/random, to generate more bits than LRNG accumulate, an attacker can block other users and successfully perform a denial of service attack.

A possible remote denial of service attack would be to open many TCP-connections against the host. For each connection, a TCP-syn-cookie is generated, which requires 128 bits, generated by the less secure interface, /dev/urandom. The urandom pool is fed new entropy from the primary pool and since the primary pool block the user if its entropy is low, it is possible to perform a successful denial of service attack remotely.

Solution. Limiting the usage per time unit reduces the possibility of a denial of service attack against the LRNG.

Falsely fed the LRNG. Since the LRNG adds the accumulated entropy to the secondary pool when the primary pool is full, a possible attack against the LRNG is to falsely fed it and directly affect the generator's output.

Solution. Separating the input and the output in the LRNG by flushing the primary pool when full, make it harder for an attacker to affect the output.

Guessable passwords. The initial usage of a rebooted system is usually for the user to provide a password, or a username- password pair. The problem arises from the fact that the LRNG use keyboard type-values as entropy. Diskless systems, as mentioned in 5.3.3, is unable to restore the LRNG from a saved state and set by a predictable function. This means the first state of the LRNG might contain the user password and since the initial state is predictable it is possible for an attacker to brute-force the password by comparing to the initial output of the LRNG.

Solution. Entropy collected from keyboard can be replaced by not using type-values, rather the timings of an event.

State reveals previous output. Mentioned in 5.3.1, the LRNG computes the output after updating its state.

Solution. By computing the output before updating the state, the LRNG satisfies the forward security requirement and an attacker cannot compute the last output by using the state of the LRNG.

Initialized state predictable. Mentioned in 5.3.3, the LRNG becomes severe weakened if the initial state is predictable.

Solution. The design always force the implementation to save the state before reboot, to be used after reboot.

The creators of Yarrow-160, Schneier and Fergusson was aware about these problems even before

the analysis was released, since they pulished an improved design principle to accompish even safer PRNGs 2003. This design is called Fortuna. To get an understanding in why Fortuna is an improvment of Yarrow-160, it is necessary to understand how it works.

6 Fortuna design: Yarrow-160 improved

Fortuna is the successor of Yarrow-160 and is devised by Bruce Schneier and Niels Fergusson. It was published in their book Practical Cryptography, *(later renamed Cryptography Engineering)*, released 2003, as an improvement of Yarrow-160.

Fortuna is not as incorporated in operating systems as Yarrow-160 but Microsoft Windows uses the Fortuna design in their standard PRNG. Patches exist[19] for most operating system to incorporate it.

6.1 Components

The Fortuna design has many similarities to Yarrow-160, and the structure can be divided into three independent components[16].

1. A **Generating Mechanism** taking a fixed-size seed and generates arbitrary amounts of pseudorandom data.

2. A **Entropy Accumulator** collecting and pooling entropy from various external non-deterministic sources and reseeds the Generator.

3. A **Seed File Control** which ensures the PRNG can generate output even if the system just been booted.

The **Generating Mechanism** convert a fixed-size state to arbitrary long outputs by using a AES-like block cipher. The internal state of the generator is a 256-bit block cipher key and a 128-bit counter, meaning the generator is basically a block cipher in counter mode or a so called stream cipher. To prevent the system if the state is compromised, the generator generate an extra 256-bit pseudorandom sequence to use as the next key for the block cipher. To ensure the outcome is statistically random, Fortuna limits the maximum size of any request to 2^{16} blocks. This increases the forward security, a documented problem in Yarrow-160, mentioned in 5.3.4.

The **Entropy Accumulator** collects real random data from various unique sources to use when reseeding the PRNG. As in Yarrow-160, such sources can be timings in keystrokes, mouse movements, mouse clicks and responses from disk drives, printers. There is no problem if an attacker can predict or copy the collected data from a couple sources, as long it is not all of them.

To be able to reseed there has to be enough events stored in a pool, that an attacker cannot enumerate the values since it has been destroyed. Fortuna accumulates entropy into 32 pools, $P_0, P_1, ...P_{31}$, compared to Yarrow-160's two and distributes them evenly in a cyclical fashion. The generator is reseed every time P_0 is long enough, and using the reseeds identification number r, pool P_i is included in the process if r is a divider of 2^i and then cleared. This prevents the mechanism if an attacker falsely fed the system, mentioned in 5.3.4, and if compromised makes the system to recover fast.

The **Seed File Control** ensures the mechanism is useable after a reboot. The seed file is a secret separate file containing collected entropy. After a reboot the PRNG reads the file to regain stability, then clears and rewrites the file. To ensure the security of the file it is necessary for the hardware and the operating system to support fully atomic and permanent file updates. This prevents the initial state to be predictable, mentioned in 5.3.4.

6.2 Comparison to Yarrow-160

The main design flaw in Yarrow-160 compared to Fortuna was the entropy estimation, mentioned in 5.3.2, used to determine when to reseed, and the problem to get it right in all situations. An attacker can falsely feed and affect the estimation to force the PRNG to reseed premature, making the system reseed the mechanism with less information, so an attacker's gathered information is not destroyed and thereof could compromise the system. This is solved by Fortuna's simple but brilliant partition

part when reseeding.

As it can provide security after reboot, the seed file in Fortuna also can imperil the security of the system and be used by an attacker maliciously. Compared to Yarrow-160 which make the client wait until the PRNG has regain stability. For each operating system incorporated in there is a need for an investigation of the particular platform to ensure the security.

3 March 2014 was the first analysis published based on the Fortuna design by Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. The analysis provides some theoretical modeling for entropy collection and stated the Fortuna design to be very good, but not optimal.

Even if the PRNGs are designed to provide high security, there is a need to verify they actually provide it. How verification of PRNGs is performed is discussed in the next chapter.

7 Verifying a random number

The importance of random numbers in computer science has been discussed as well as various different ways of creating sequences of pseudo random numbers. However the randomness used, especially in information security, needs to hold a certain standard. The security may very likely otherwise be compromised. It is therefore important when choosing or designing a random number generator, either a PRNG or TRNG, that the randomness can be verified in some way. This can be done as discussed further on using standard test, common criteria or certifications. The different approaches used vary depending if the number has been created by a PRNG or a TRNG. [11]

7.1 Standard test

Random number generators can be tested in different ways. The test methods can be divided in to three different areas, statistical, which is most used, transformation and complexity. A mix of the three methods can also be used. Which tests that is suitable for carrying out depends on if a PRNG or TRNG is tested. There is no test that is defined to be complete, and most often more than one test is carried out for determining randomness. The tests are constantly improved and therefore the quality of random number generator is increased. This is because it sets more stringent requirement on the generators to pass the tests. This is however needed since not only the test are getting more sophisticated but also the attacks on random number generators is getting more advanced. [12]

7.1.1 X^2-test

The X^2-test is a classical statistical test. The expected distribution pattern of output data creates a null hypothesis. The null hypothesis is then compared with the distribution pattern of the actual output. If the difference of the two sets of data, X^2, is lower then a predefined limit the hypothesis hold and the data is chi-squared distributed. The sum X_s^2 is calculated as follow:

$$X^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

O_i is the observed result, E_i is the expected result and n is the sample size. [10]

The chi-square test can be used to determine the statistical properties of both PRNG and TRNG[11]. This test is also the foundation for a lot of other tests regarding randomness, example the Diehard test discussed in the next section. When this test is used for verification of a PRNG the null hypothesis often is that the PRNG is a random number generator that generates uniform random numbers.

7.1.2 Die-hard test

The mathematician George Marsaglia developed another statistical test for randomness in 1995. It is named the Diehard Battery of Test of Randomness and is composed of several smaller independent tests. They examine if they random number generator is good enough to pass the test as a whole. It can be used for TRNG, but it is mostly used for PRNG. The smaller test are evaluated with a p-value from 0 to 1, where a 0 value indicate that the test was successful in generating a random value. [11]

7.1.3 Transformation test

A test for verifying randomness can be designed using transformation. One of the first to apply this

was the mathematician Subhash Kak. He measured the randomness in a sequence of random numbers using the Walsh-Fourier transformation.

"A sequence shall be said to have no pattern or be random if the number of independent amplitudes in the Wash-Fourier transform is equal to the length of the sequence itself, i.e., $2k$." - Subhash Kak

Using the Discrete Fourier Transform on a binary sequence is done as follow according to NIST. [14]

1. Each binary input, e_i is put in the equation: $x_i = 2e_i - 1$
2. A discrete Fourier Transform is then applied on $X = x_1, x_2, \ldots, x_n$. A complex sinusoid is then created, $S = DFT(X)$, which characteristics depend on the frequency and length of the input.
3. A subsequent of S, S' is taken and calculated $M = modulus(S')$. This produces a series of peaks.
4. A value, T is calculated using a predefined threshold value.
5. The number of peaks above the value T is calculated as N_0 and the number of peaks below T as N_1.
6. This to numbers are then used in calculating d

   $$d = \frac{(N_1 - N_0)}{\sqrt{n(.95)(.05)/4}}$$

   Where n is the length of the bit string $e = e_1, e_2 \ldots, e_n$. The value of d will be to low if the distribution of peaks below and above T is too uneven.
7. A value p is the calculated: $p = erfc(|d|/\sqrt{2})$, if the value of $p \geq 0.01$ the input sequence is random.[12]

7.2 Certification

Deciding if a PRNG or TRNG is producing sufficiently random numbers can be done using tests as described. Though it exist easier ways to verify the randomness by considering what more knowledgeable parties regards as sufficiently random.

One organization whose opinion about computer security is highly regarded is NIST, National Institute of Standards and Technology. They are more thoroughly discussed further on. Another approach is to look at what is currently used most common by others and hold this as a standard. This is what is done when PRNG is classed as a CSPRNG.

7.2.1 CSPRNG

A CSPRNG, Cryptographically secure pseudo random number generator, is a PRNG that is used for cryptography. Not every PRNG can be used for cryptography since the security demand is rather hard. This is because the CSPRNG must be able to resist cryptanalysis, since this is always a threat against cryptography. For a CSPRNG it is extremely important that the seed is kept secret. This is because the PRNG in cryptography create for example keys for encryption or nonce/IV for block ciphers. If the random number is not random enough and an attacker then can determine the next number, the attacker can also gain the key or the starting point of the cipher and then the encryption will not be secure[26].

7.2.2 NIST

NIST, National Institute of Standards and Technology, belongs to United States department of Commerce. It has six different laboratories and the Information Technology Laboratory (ITL) is one of them. One of the areas covered by ITL is computer security. Therefore a lot of research for creating test and tools in cryptography is made, and cryptography is as previously discussed dependent on strong randomness[20].

NIST offer many different test suits for randomness. They also define the standards regarding information security. This includes many things, examples are which hash function or encryption algorithm to use. NIST also decide when a standard is out of date and a new one is needed.

There recommendations of standards are often followed, voluntarily or involuntarily in preprogrammed applications. They have however got critics which do not like that they are an organization linked to the authorities. The critic has increased after it has been revealed that one of the standards regarding a recommendation for random

number generator that contained a backdoor used by the NSA[21]. NIST have very recently removed there recommendation regarding this specific random number generator[22].

8 Attacks

Even if the random number generator is a CSPRNG or recommended by NIST it can still be the focus of an attack. The chance of it succeeding is however then smaller comparing to using a weaker PRNG. The following part of the report will be concentrating on different attacks on PRNGs.

It exist several attacks on a pseudo random number generator and they can be divided in to three different groups depending on their characteristics. The groups are Direct Cryptanalytic Attack, Input Based Attack and State Compromise Extension Attacks and they are all discussed below.

The attacks have in common that they aim at being able to distinguish between an output from a PRNG and a random output. Thereafter the attacker wants to be able to determine the next number to be generated or figure out previous numbers. This can be done by either generating the input and therefore changing the entropy or by knowing the entropy.

If an attacker can determine the next number to be created by a pseudo random number generator the generator is no longer secure. The security of the system that the PRNG is part of is then compromised even if the rest of the system is secure the system as a whole is not. It is therefore important to have a PRNG that can withstand attacks.

8.1 Direct Cryptanalytic Attack

This attack is when an attacker is able to directly distinguish between an output from a PRNG and a random output. The attack can be carried out on most PRNGs but not the one where the output is not shown as for example when triple-DES keys are generated.

8.2 Input Based Attack

This attack is when an attacker knows or can control the input to the PRNG and thereafter cryptanalyze the output. The attacker can then remove existing entropy of the PRNG and replace it with a known inputted state. The attack can be divided in three different groups depending on what is known about the input[23].

Known input. Some of the characteristic of the input example the length is known to the attacker.

Chosen input. The input can be a message such as a password or pin-code and this comes into the PRNG as entropy samples.

Replayed input. This is similar to the chosen input but the attacker chooses to put the same input many times.

8.3 State Compromise Extension Attacks

This attack tries to take advantages of temporary security breaches for example an inadvertent leak. The attack is achievable because the eternal state S becomes known. This can be done when the generator starts up again and therefore the entropy becomes insufficient and the attacker can make an initial guess of S. Knowing the state S can give the attacker information about future output or being able to recover previous output. The attack can be further divided in to four different approaches:

Backtracking Attacks. This is when the state S is used to figure out the previous output of the PRNG.

Permanent Compromise Attacks. This attack is permanent because then all future as well as all previous output can be determined when S is known.

Iterative Guessing Attacks. The attack uses the knowledge of state S at time t and the PRNG outputs over the time T and thereafter guesses the state S' at time $t + T$.

Meet-in-the-Middle Attacks. This attack is a combination of the previous two, Iterative Gussing and Backtracking. Knowing the state S at time t and state S' at time $t + T$ also allow the

attacker to figure out the state $S*$ at $\frac{t+T}{2}$[20].

8.4 Known real attacks

There have been several examples of attacks against pseudo random number generators, which have succeeded. They have resulted in that the security of the system has been compromised. Often is the underlying cause insufficient entropy, which makes it possible for the attacker to guess the initial state.

Playstation3, Microsoft Windows, OpenSSL and Netscape are just some examples of products that have had deficient PRNGs and have therefore been able to be attacked.

In the case of Netscape it was an early version of the web browser that used a PRNG to create a SSL, Secure Socket Layer. The PRNG used three variables as a seed: the time of day, the process ID, and the parent process ID. Neither one of these three are hard to figure out for a determined attacker. It was therefore classed unsecure and got corrected to later versions[24].

OpenSSL up to versions before 0.9.8g-9 on Debian-based operating systems had one important but difficult-to-understand source code line commented out, which led to insufficient entropy gathering and to concrete attacks on TLS and SSH protocols[25].

NSA was claimed of implementing a backdoor in the NIST-recommended CSPRNG, Dual Elliptic Curve Deterministic Random Bit Generator, or Dual_EC_DRBG. The backdoor was found in 2007, but was recommended by NIST until 2014, after Reuters reported on the existence of a $10 million deal between RSA and NSA to set Dual_EC_DRBG as the default CSPRNG in BSAFE and a public comment and review.

9 Conclusions

Due to the vitality of randomness in computer science and information security, the importance of an unpredictable creation of random sequences is clear. Many systems rely on the unpredictability and when it is compromised, the systems become vulnerable, making the PRNGs attractive targets.

As the PRNG more and more rely on non-deterministic information within the deterministic environment to create more secure and better pseudo random number generators, it is obvious it never possible to create true randomness. So when using systems with higher security levels it is recommended to use TRNGs.

The design principles of Fortuna, compared to Yarrow-160 is recommended in a security manner, due to their simple but brilliant idea of partitioning the entropy into multiple pools and use it in different rates. It would not be surprising if the Fortuna design appears in operating systems default PRNGs in the future.

When reviewing the open source code of the LRNG, the lack of documentation in the code is noticeable compared to other files in the open source project. Since the ability to review source code is a vital part of the development in open source projects, the PRNG would benefit by being documented, and avoid security by obscurity.

The output of a PRNG can be tested, in different ways, to verify randomness. This is however time-consuming and a faster way is to listen to recommendation from more knowledgeable parties. One organization that has a lot of influenced over recommendations is NIST.

There is always the threat of attacks against a random number generator. It exists many different kinds of attacks and they can be divided into different groups depending on there characteristics. The characteristic can be what kind of information the attacker aim to get or the approach of the attack. It is always important that the randomness used in a system holds a certain standard. Otherwise the security of the system may very likely be compromised

References

[1] John von Neumann, 1951, *Various techniques used in connection with random digits.* Monte Carlo Method, National Bureau of Standards Applied Mathematics Series vol. 12 (Washington D.C.: U.S. Government Printing Office 1951): pp. 36-38,

[2] Ivar Ekeland, 1996, *The Broken Dice, and Other Mathematical Tales of Chance.* University of Chicago Press, ISBN 978-0-226-19992-4.

[3] Gregory J. Chaitin, 1975, *Randomness and Mathematical Proof.* Scientific American 232, No. 5 (May 1975), pp. 47-52,

[4] A. N. Kolmogorov, 1965, *Three approaches to the quantitative definition of information.* Problemy Perdachi Informatsii, Vol. 1, No. 1, pp. 3-11, 1965,

[5] David J.C. MacKay, 2003, *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press 2003, Vol. 1, No. 1, pp. 2, 1965,

[6] Pierre L'Ecuyer, 1975, *Pseudorandom Number Generators.* Encyclopedia of Quantitative Finance, R. Cont, Ed., in volume Simulation Methods in Financial Engineering,

[7] Oracle USA, 2011, *java.util.Random.* Java SE Developer Documentation,

[8] Oracle USA, 1995-2007, *java.util.Random 6b14 source code.* Java SE Developer Documentation,

[9] Jiri Fiala, Vaclav Koubek, Jan Kratochvil, 2004, *Mathematical Foundations of Computer Science.* 29th International Symposium, MFCS 2004 Prague, Czech Republic, August 22-27, 2004 Proceedings.

[10] John H. McDonald, 2009, *Chi-square test for goodness-of-fit.*

[11] T.Nijm, 2002, *Slumptalsgeneratorer för säkerhetssystem*,

[12] Andrew Rukhin , 2010, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*,

[13] Terry Ritter , 2002, *Randomness Tests: A Literature Survey*,

[14] NIST , 2010, *National Institute of Standards and Technology*,

[15] Bruce Schneier, John Kelsey, and Niels Ferguson, 1999, *Yarrow-160: Notes on Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*,

[16] Bruce Schneier and Niels Ferguson, 2003, *Cryptography Engineering: Design Principles and Practical Applications*,

[17] Apple Inc., 2010, *Apple Inc. Open Source Browser*,

[18] Apple Inc., 2012, *Apple Inc. iOS Security*,

[19] Jean-Luc Cooke, 2005, *Fortuna Random-2.6.12-rc2-mm3.patch*,

[20] Bhatnagar, V Cheruvu,C, 2007, *Pseudo Random and Random Numbers*,

[21] Kim Zetter, 2013, *How a Crypto 'Backdoor' Pitted the Tech World Against the NSA*,

[22] Larry Seltzer, 2014, *NIST finally dumps NSA-tainted random number algorithm*, .

[23] Bruce Schneier, 1998, *Cryptanalytic Attacks on Pseudorandom Number Generators*,

[24] Goldberg, I. Wagner, D, 1996, *Randomness and the Netscape Browser*,

[25] National Vulnerability Database, 2008, *Vulnerability Summary for CVE-2008-0166*,

[26] Carl Ellison, 2007, *Cryptographic random numbers*,

[27] Zvi Gutterman, Benny Pinkas and Tzachy Reinman, 2006, *Analysis of the Linux Random Number Generator*,