On Measuring Randomness

Karel Lang 37754
Master's thesis in computer science
Supervisor: Jan Westerholm
Faculty of Science and Engineering
Åbo Akademi University

Table of Contents

Abstract

Is it possible to determine what randomness is let alone measure and classify it? Can random number generators be tested and then compared with each other? How can we know if a randomness test is useful? These are questions that are thoroughly studied in this thesis.

One of the main goals of this thesis is to explain interesting details that can be found in randomness and to study how pseudorandomness compares to real randomness. Alongside this paper, a program will be created in order to study and measure pseudorandom number generators and randomness tests. The program will allow the combination of random number generators and randomness tests in order to study any hidden similarities of different generators and tests. The classification of randomness and randomness tests will also be studied.

The first chapter reviews the more philosophical discussions concerning randomness and its types. The second chapter is slightly more practical, as it explains in which ways randomness can be measured and defines what a good pseudorandom number generator should look like. The second chapter also reviews concepts and terminology which are crucial in order to understand later chapters. The third chapter studies randomness tests and tries to define an ideal randomness test. The fourth chapter explains the software that is created along with this paper to study randomness tests and pseudorandom number generators and the fifth chapter shows the results of experiments done with the software.

1 Introduction

Randomness offers unique concepts for study. It is a purely mathematical concept, as it can be defined precisely within specific axioms, yet the act of measuring randomness (or the lack of it) can only be done through statistical methods, the same methods that are used in applied mathematics. It is rather interesting that some concepts within mathematics need to be dealt with in the same manner as observations of the real world. Using statistical methods on perfectly defined mathematics sounds like a contradiction, as one can only obtain an approximation from statistics, yet it is required in order to study randomness. Randomness by definition cannot be studied analytically, since it cannot even be generated with a function.

Randomness is useful in almost all sciences. Examples include simulations, quantum mechanics, realistic terrain generation, cryptography and much more. Where statistics is needed, randomness is often also needed. Since randomness is used often, researching it should be of at least some importance. This is where random number generators (RNG), pseudorandom number generators (PRNG) and randomness tests appear.

As creating true randomness is difficult (if not impossible) the use of PRNGs is popular. These PRNGs are algorithms which create seemingly random numbers, but they are completely deterministic. In most, if not all use cases, the use of a PRNG is preferred over the use of a true RNG due to it being much faster for generating random numbers. It is also important that the ‘randomness’ of the PRNG should be as similar to real randomness as possible. The first PRNGs were simple and fast but the output they generated was of poor quality. Some examples are the Middle-square method [1] from 1951 or possibly earlier, Lehmer random number generator [2] from 1951 and the Linear Congruential Generator [3] (LCG) from 1958. The LCG was thought to be a very good PRNG as it passed all randomness tests but if values generated from an LCG are mapped into two or higher-dimensional plots, the points will always fall into planes. A randomness test called the Spectral test [4] was created just to study this behavior of LCGs in order to choose better parameters for LCGs, as they were otherwise still good enough for most applications requiring random numbers. Many applications require random numbers of better quality than what any LCG can offer. Thus, the development of new PRNGs has not died out. It is an arms race between PRNGs and randomness tests, which will most likely never end. This makes randomness so fascinating and is part of the reason why I decided to write my paper about measuring randomness.

1.1 Randomness and Quantum Randomness

For a commonly used word, defining ‘random’ is surprisingly difficult. For starters, something that can be predicted is not random. Something that cannot be predicted would mean that there is no inner process to it and no inner process would mean that it lacks the cause-and-effect relationship. Therefore, randomness is above cause-and-effect. This is a good but not a perfect definition, as will be later explained in chapter 1.2.

The day-to-day use of the word random consists of anything that we perceive to be random, such as implausible events and rare occurrences. This is a fallacy, as the outcome of a random event is no longer random. Even if we could say that the outcomes themselves are ‘random’, there is no such thing as different amounts of randomness. Winning in the lottery is as ‘random’ as not winning in the lottery, only the probability is different. Things that are deterministic but too complicated to predict perfectly are also perceived as random, like the weather or a coin toss. As such, the common use of the word random has very little to do with the scientific definition of it. Shortly explained, the common use of the word random is for outcomes which are unlikely to happen and/or deemed odd. What, then, is the scientific definition?

As with many fundamental properties of different sciences, the study of these properties is akin more to philosophy than science, since the questions about these properties are often impossible to answer. Examples of fundamental properties include causality in physics, axioms in mathematics, computability in computer science and randomness in probability theory. Probability theory uses randomness but how does it describe it? This will be studied chapter 1.2.

Even if we had an exact definition of randomness, this does not say that true randomness exists in the real world. Current research in quantum mechanics does actually suggest that true randomness exists [5]. There is an ongoing debate concerning the deterministic or nondeterministic nature of the universe. However, it will still be impossible to prove that true randomness exists in the quantum world, as hidden variables (of some level of complexity, which might as well be above human understanding) would always be a possible explanation for the perceived randomness found in quantum mechanics. There is also the possibility of someone inventing a test which would render it impossible to explain the behavior with hidden variables. There is a multitude of possible outcomes in the struggle to prove or disprove true randomness in quantum effects. Below are the three different outcomes that I think are most likely to happen if we will ever advance in this matter.

1. A test is made which is then proven to render the existence of hidden variables impossible.

2. We prove that such a test as explained in 1 is possible to make, but it will always remain a non-practical test.

3. We prove that such a test cannot be made.

Of these three outcomes, the third one is most likely, as such a test seems implausible since it feels similar in scope to the halting problem. Note that true randomness exists only if outcome one is true. Even if outcome two is true, it does not mean that the result of the test is one or the other.

1.2 True Randomness and Axioms

Which are the basic properties of true randomness? How can we use the abstract concept of true randomness in mathematics? In order to understand how randomness is used, we need to look at the basics of probability theory, its axioms. Here are three of the five Kolmogorov axioms [6].

Axiom 1: $P(A) \geq 0$, for all $A \in F$

Axiom 2: $P(\Omega) = 1$

Axiom 3: $P(A \cup B) = P(A) + P(B)$ for all $A, B \in F$ such that $A \cap B = \emptyset$

Ω is the sample space set, F is the event space set (powerset of Ω), P is the probability measure and $\emptyset$ is the empty set. Note that in his book, Kolmogorov specifies five axioms, but the first two are often left out since they are definitions rather than axioms (the second axiom simply states that F contains the set Ω).

Interestingly, randomness is not defined in the axioms. Actually, the concept of randomness is not even mentioned in the axioms, only probability is mentioned. This is because probability theory does not need the concept of randomness to work. The system built on the three axioms creates a branch of mathematics which can be seen as separate from randomness in the same way as the Peano axiom of addition is separate from counting. The scientific definition of randomness could then be something that satisfies the three Kolmogorov axioms.

The idea is that the Kolmogorov axioms are not the same as randomness, but they describe the behavior of randomness perfectly. The Peano axiom of addition defines exactly that, addition. Counting is not the same as addition, but it can be precisely defined by addition.

Note how this randomness that is tied to the three Kolmogorov axioms is above cause-and-effect, but in an extremely precise way. Is it possible to define a type of randomness that is above cause and effect but does not follow the Kolmogorov axioms? What sort of randomness would that even be, and do we have any use for it?

Quickly glanced at the concept it seems that such randomness simply cannot exist. For example, axiom 1 specifies that the probability of an event occurring must be larger or equal to zero. If we were to break this axiom and claim that something has negative probability, what would

that even mean? Interestingly, negative probabilities have been studied since the 1930s [7] because they seem to appear in, not surprisingly, quantum mechanics [8]. This led to the definition of the quasiprobability distribution which allows the use of negative probabilities [7]. Note that in the paper, the quasiprobability distribution does not yet have a specific name but is instead mentioned as a wave function. However, even if there is some use of negative probabilities in quantum mechanics, one could still ask, whether such a thing truly exists or whether it is only a neat way of calculating something that would otherwise be too difficult to work with. This will inevitably lead to the question of whether mathematics itself truly exist. Do negative values exist, what does it mean to have a negative number of apples, let alone an imaginary number of apples? Thus, the question of how real negative probabilities are will be left alone.

If axiom 1 can be left out, what about axiom 2 and 3? Even if we were to deal with negative probabilities, this would break neither of the two remaining axioms, not to mention that breaking the two remaining axioms would be even more exotic compared to the first one. Imagine that you are throwing a die; if the probability of obtaining an outcome would not be 1, what would the remaining part be? If this remaining part were the probability of not obtaining an outcome from throwing the die, one could add this to the probabilities as it would also be an outcome. Would not the chance for that happening be zero? Thus, we would immediately have a paradox if we were to not follow axiom 2. However, should not choosing 1 to represent absolute certainty have to be an arbitrary decision? Thus, if we were to say that absolute certainty would be represented by 2, what would happen concerning the mathematics of probability theory?

If we have a fair coin, the chance of obtaining heads would be 1 and for tails it would also be 1. Thus, the chance of obtaining two heads in a row would be 1 * 1 = 1, but what would that mean? We know that the chance should be one fourth. Furthermore, if we were to sum the sample space set according to axiom 3, we would obtain 4 which is more than what axiom 2 allows as absolute certainty should be 2. What went wrong? In fact, we made a mistake in the calculation, as we did not take into account that absolute certainty is 2. Our calculation should actually have been (1 / 2) * (1 / 2) = 0.25 to obtain the correct answer. However, the value 0.25 is correct only when $P(\Omega) = 1$, so what happened here? As we did the multiplication, we actually calculated “one of two” times “one of two” which would then be “one of four”. Since $P(\Omega) = 2$, though, we need to change “one of four” into “x of two” to obtain the correct value. Thus, by dividing by two we obtain the correct value, which is 0.5. All ratios remain the same if $P(\Omega) = 2$, but the calculations are slightly more tedious. Thus, $P(\Omega) = 1$ is not an arbitrary decision, but it is used because it is the easiest to work with.

Overall, it seems that while working with randomness through probability theory, the calculations do not require a definition for true randomness to work, but instead something that

seems to be similar to true randomness, until it is not when it comes to the quasiprobability distribution. Thus, the concept of true randomness becomes even more difficult to define. One could say that true randomness is something that we can only define by limiting what it is not. That is to say, true randomness lies within the Kolmogorov axioms but its truest nature, about not following cause and effect will always remain a mystery.

1.3 Types of Randomness

What does one mean by types of randomness, should something not be either random or not random? This is surely true, when looking from the cause and effect point of view. Lumping everything that is above cause and effect into the same group may be counterproductive. Furthermore, what about randomness that is tied to cause and effect, can we even say that such a thing is random? According to our definition at the beginning of chapter 1, it would not be random, but something that is deterministic yet behaves as if it were random could still be useful. This deterministic "randomness" is called pseudorandomness and it is useful in many sciences. It is the randomness or lack of randomness of pseudorandom number generators that we mostly want to measure with the use of randomness tests.

A pseudorandom sequence is a sequence that appears to be random but actually follows some pattern. The term "appears to be random" is unscientific, as it cannot be defined, and as such, any possible sequence that is not random can be seen as pseudorandom. Thus, everything that generates outcomes can be placed in one of the two following boxes: random generators and pseudorandom generators. The incremental algorithm is also a pseudorandom generator, albeit an inferior one. Why is it inferior and can this "inferiority" be scientifically determined? Note that for all scientific research, the results are never completely certain, meaning that we will require some sort of cutoff point where we can conclude that a result is clear enough to claim it to be true. This cutoff point for certainty is completely arbitrary and can be compared to the arbitrariness of when a deterministic sequence is deemed to be random enough to be called pseudorandom. This is one of the main questions that this thesis tries to answer, and the detailed analysis for this question can be found in chapter 2.

We can divide randomness first into randomness and pseudorandomness. Randomness can be further divided into definable and undefinable randomness. In order to understand what these types of randomness truly are, we need to start from the lowest level. This would be the simplest random generator. Note that we will be using numbers as the output to make the explanations easy

to understand.

What is the simplest random number generator? This would arguably be the generator that generates a discrete uniform distribution consisting of two numbers: zero and one. The likelihood of a zero is the same as a one, and it follows the Kolmogorov axioms. We will call this random number generator R_0. One could argue that the continuous counterpart would be better since the discrete form can only approximate a continuous value. However, infinite precision behaves in unintuitive ways. Even if such precision could exist in the probabilities of half-lives and other quantum-related randomness, it would be impossible to store the number since infinite precision leads to infinite data and, thus, the discrete version is arguably more useful.

Note that the concept of randomness can be understood even without defining a specific probability distribution. However, in order to make use of randomness, a probability distribution is required.

With the random number generator R_0, it is possible to create almost any probability distribution. This can be achieved by using a Turing machine to transform the output. Let us call the set of all random number generators that can be imitated with our random number generator "definable randomness". Below are some examples.

We define the random number generator as the function R_0, where R_0 will give as output a 0 or a 1. We now want to create the continuous counterpart of R_0, which we will call R_1. R_1 will have the output range [0, 1) (where [is the closed interval and) is the open interval). Since R_1 is continuous, we can only approximate its probability density function, but we can do so to any precision we want.

$$R_{0\_1}(N) = \sum_{k=0}^{N} R_0 * 2^{-k}$$

$$PDF(R_1) = \lim_{N\to\infty} PDF(R_{0\_1}(N))$$

where PDF is the probability density function, lim is limit and $\sum$ is summation. Note that PDF is a function that runs the function that is given as input an infinite number of times and creates the probability density function from that information.

We could split definable randomness into continuous and discrete definable randomness as creating R_1 with the help of R_0 with infinite precision is a supertask. However, this would also mean that we could not do anything with R_1 due to R_1 requiring infinite memory in order to handle its output. A Turing machine is not capable of dealing with infinite data and, thus, it is more useful to define imitation as obtaining any desired level of precision instead of using infinite precision.

R_0 can also create any type of discrete distribution. Having an equal probability to obtain a 0, 1, 2 and 3 is trivial, since we can combine two R_0 to produce this distribution. We set [0, 0] = 0, [0, 1] = 1, [1, 0] = 2 and [1, 1] = 3. We call this R_2 and its PDF can be created in the following way:

$PDF(R_2) = PDF(R_1 * 2 + R_1)$

If we want to create an equal probability of obtaining a 0, 1 and 2, the process becomes more complicated. One method would be to use the following method:

$R_{0\_2} = R_1 * 2 + R_1$

R_{0_3} = If R_{0_2} is 3, generate a new R_{0_2}, otherwise return it.

$PDF(R_2) = PDF(R_{0\_3})$

The main problem with this approach is that we cannot determine how long it would take for a Turing machine to run this algorithm. If we are unlucky, R_{0_2} could generate 3 multiple times in a row, at worst, an infinite number of times. Therefore, another method must be found.

Is there another method? Each time we use R_0 we will obtain one of two outputs, and no matter how many times we run R_0 we will always create a power of two number of possible outputs. No matter what integer power of two we have, it is never divisible by three, meaning that creating a perfect method is impossible. However, we can always approximate the output of R_2 to any precision we want with the help of R_1:

$PDF(R_2) = PDF(Floor(R_1 * 3))$

where Floor is the floor function. The floor function rounds a given value down to the nearest integer. Since we already know how to create R_1 with the help of R_0, we can now conclude that this is a valid way to approximate R_2 using R_0.

Now that we have defined what "definable randomness" is, the concept "undefinable randomness" can be explained. The key difference is that in undefinable randomness the probability distribution is incomputable. A simple example of a random number generator that would generate undefinable randomness would be R_U:

$R_U = 1$ with the probability of Ω_U

$R_U = 0$ with the probability of $1 - \Omega_U$

where Ω_U is Chaitin's constant [9] for some universal Turing machine U. A universal Turing machine is a Turing machine which takes as input both a Turing machine and the input of the Turing machine. $PDF(R_U)$ would be undefinable, since Ω_U is an incomputable constant. We know that Ω_U must be within]0,1[[10]. Therefore, Ω_U can be used to create a probability distribution for R_U.

What makes this even more mind-boggling is that Chaitin's constant itself is random [9]. Thus, the random number generator R_U is a random number generator with a random probability distribution. This means that we could create the algorithm $R_\Omega(s)$:

$R_\Omega(s)$ = the s:th bit in Ω_U

$R_\Omega(s)$ has the state s, meaning that it is a random number generator with a state and its probability distribution is also random, or is it? What can we know about $PDF(R_\Omega(s))$? Interestingly, even if

we cannot calculate Ω_U we know that it is a normal number. This means that PDF ($R_\Omega(s)$) is equal to PDF (R_0). This is because a random decimal must, by definition, be normal.

1.4 Explaining the Random Sequence

Understanding what a random sequence is and what it is not is a daunting task due to all the semantics that needs to be fully understood, yet it is crucial to fully comprehend random sequences. In the paper "Randomness: quantum versus classical" [11], Andrei Khrennikov aims to explain the basics of the classical theory of randomness and compare it to quantum randomness. One of the goals of the paper is to demonstrate why it is necessary to distinguish between probability and randomness. In chapter 2 of his paper, Khrennikov explains how some sort of "formal theory of randomness and its interrelation with probability has to be developed" [4.5 page 3]. He further states, as an example, that a sequence of bits which appears random could be the first digits of pi in binary form. Generating a sequence of one hundred bits only to receive a string of ones does not seem random, yet it has the same chance of occurring as any other of the 2^{100} outcomes that exist.

It is easy to understand how one can claim that one sequence appears to be less random than another, but this whole concept simply amounts to predetermined randomness tests and retrospective randomness tests. In the same way as all numbers have some unique property to them, so do random sequences. There is a countably infinite number of deterministic algorithms which could create the same finite sequence of numbers. Thus, one could always invent an algorithm for generating a finite sequence which makes the finite sequence appear nonrandom. This is, however, only after the fact. If we were to come up with the deterministic algorithm in advance, it would be far more surprising. If we in advance create some binary sequence of length one hundred and the apparently random sequence that is then generated matches the sequence, it could, in some sense, be seen as unlikely as a string of ones. However, if that sequence were generated without it being specified in advance, the sequence would be unremarkable.

I aim to claim that determining if a finite sequence appears to be nonrandom should only be done with randomness tests that have been chosen before seeing the sequence. In other words, without randomness tests, all possible outcomes are equally 'random'. The one hundred sequential ones is not a special outcome if we have not predetermined what counts as a special outcome. Only with context can we claim that an outcome is unlikely. The Kolmogorov complexity algorithm, which is mentioned at the end of chapter 2 of Khrennikov's paper, is simply a randomness test, albeit slightly too effective for measuring the quality of PRNGs, not to mention that the algorithm is

also incomputable. The Kolmogorov algorithm will be discussed in detail in chapter 3.1.

Where this logic will not work are the infinite sequences. The chance of a random infinite sequence to appear nonrandom is zero. This can be proven by thinking of an infinite random sequence as a real number. As Cantor's diagonal argument proves, the set of real numbers is uncountably infinite. If a sequence appears nonrandom, there must be a way to prove that it is nonrandom, meaning that an algorithm to verify this must exist. Since the set of all programs is countably infinite (the set of all natural numbers is countably infinite, and each program can be seen as a natural number), we can conclude that the probability of having an infinite sequence appear nonrandom is zero. We can also conclude that the Kolmogorov complexity of a nonrandom infinite sequence must always be finite, as an infinite program cannot be executed and therefore is incomputable.

Chaitin himself remarks in his paper "A Theory of Program Size Formally Identical to Information Theory" [12] on page 19 concerning random strings (which can be seen as sequences): "In the case of infinite strings there is a sharp distinction between randomness and non-randomness. In the case of finite strings it is a matter of degree", where the method for measuring non-randomness is in fact Kolmogorov complexity (which is called program-size complexity in his paper).

2 Measuring Randomness

At first it may seem unreasonable to discuss measuring randomness. What is there to measure in true randomness? The results would never change nor would they differ from any analytical calculations. We do not need to measure true randomness, since it is already mathematically precisely defined. However, as discussed earlier we are still unsure and will always remain unsure if true randomness exists as something more than just an abstract mathematical concept. Thus, the actual objective is not to measure randomness, but the lack of it. This can be done with the help of randomness tests.

It is impossible to prove that something is random, but measuring the lack of randomness could be done with the help of statistics. Thus, the goal of a randomness test is to measure the lack of randomness in a sequence of numbers. A randomness test can be seen as a criterion that is well defined. All a randomness test does is that it checks if the given sequence of numbers meets its criteria. The measurement is done with the help of statistics. The output given by a randomness test is a variable which is as random as the input the randomness test evaluates. The output variable is a real value between 0 and 1 where the higher the value, the better the tested sequence of numbers matches the criteria. Note that an output of 1 does not mean that the sequence appears to be random. If the sequence truly were random, the output value should have a uniform continuous distribution between 0 and 1. The output value tells how likely the outcome of the randomness test is, if the data which was tested were random. Thus, the result of a randomness test should be different each time it is used to test the same random number generator. This will be explained in depth in chapter 2.2.

Even if we can measure the lack of randomness, the output values will still only be probabilities. As long as the sequence of random numbers could be generated by a true random number generator, it is still possible that the sequence truly is random. A finite sequence that a true random number could not generate cannot exist. We can always calculate the probability of a finite sequence being created by a true random number generator to be above zero. Working with a discrete uniform distribution, where the only outcomes are one and zero, we could always calculate the probability for any sequence. As long as the sequence is finite, the chance of it happening is above zero. Thus, even if we were given a sequence of ten thousand ones, we cannot be certain that it is nonrandom.

This makes measuring the lack of randomness complicated. Ultimately, we can never be sure of anything, yet we still need to conclude something. The same problem exists outside random number testing. In any branch of science where experiments are done, the results can never be certain. Due to all of the factors which influence the results of an experiment but cannot be controlled, the results of most, if not all, experiments can never reach absolute certainty. A method

called the statistical hypothesis test is used to provide a sort of confidence. The statistical hypothesis testing method is explained in the next chapter.

2.1 Statistical Hypothesis Test

In an experiment where absolute certainty cannot be achieved, a level of confidence can be used instead. This level of confidence can be represented as a sigma or p-value. Sigma and p-value represent more or less the same thing, but in opposite directions. The higher the sigma the higher the confidence, and the lower the p-value the higher the confidence. They are explained in chapter 2.2 “Sigma and p-values” in more detail. For a statistical hypothesis test, we will need to define what level of confidence we want to reach in order to accept the research data. This means that a sigma or p-value needs to be defined before performing the statistical hypothesis test.

A statistical hypothesis test consists of a hypothesis that can be tested with research data. The counterpart of the hypothesis is the null hypothesis. The null hypothesis is that the research data lack the required statistical significance to reject our current conclusions and, therefore, we should remain in status quo. Sigma and p-value can be seen as cutoff points. If we come to the conclusion that our research data remained below sigma or above the p-value, we accept the null hypothesis. If our research data were statistically significant enough to be above sigma or under the p-value, we reject the null hypothesis. Below is an example of how to use the statistical hypothesis test. This example is based on the discovery of the Higgs boson.

A team of researchers at CERN claimed that they have discovered the Higgs boson with the significance level of five sigma. Note that this is not exactly how the discovery progressed, but this explanation is sufficient as an example. The accepted significance level for such a discovery is four sigma. This means that the discovery had a statistical significance high enough to reject the null hypothesis. The null hypothesis was that they did not make a new discovery. This does not mean that the team discovered the Higgs boson, it only means that their findings are statistically significant enough to conclude that they did, in fact, discover the Higgs boson.

For all statistical hypothesis tests there are four different outcomes. See the table below.

	Data were not significant enough to reject the null hypothesis	Data were significant enough to reject the null hypothesis
Higgs boson does not exist*	The right outcome	Type I error
Higgs boson does exist*	Type II error	The right outcome

In the Higgs boson case, we are in the column ‘Data were significant enough to reject the null hypothesis’, but we will never know in which row we lie. We can only become more certain about it being the bottom row. In the table, we also see two types of errors: the Type I error where something which is not true is believed to be true, and the Type II error where something which is true is not believed to be true.

It is important to understand that the Type I error is much more harmful for scientific research than the Type II error. Experiments can be seen as building blocks, which new experiments rely on. It is also important to note that accepting the null hypothesis does not mean that we become more certain of the null hypothesis being true, it only means that the data were not statistically significant enough to reject it. The stricter the sigma or p-value we require, the smaller the chance is that we end up in the Type I error cell. The more data we use in the test, the smaller the chance is that we end up in the Type II error cell. Thus, in order to minimize the chances of making the wrong conclusion, we need as much data as possible and a strict sigma or p-value.

Note that the table contains asterisks. This is because the rows “Higgs boson does not exist” and “Higgs boson does exist” are not entirely correct. The observed outcome could occur even if the Higgs boson did not exist. Something else could be the cause for the discrepancy between the simulated data, which lacks the anomaly, and the generated data. Thus, one could change “Higgs boson does not exist” to “discrepancy does not exist” and “Higgs boson does exist” to “discrepancy exists”.

Randomness tests also use the statistical hypothesis testing method. A randomness test outputs a p-value which tells how close the sequence of numbers used as input was to reaching its criteria. The p-value is then compared to a predefined cutoff point. If the p-value is below the cutoff point, we can conclude that the sequence of numbers used as input matches the criteria of the randomness test and, thus, the null hypothesis is rejected.

Below is the statistical hypothesis test outcome table used by randomness tests:

	Accept the null hypothesis test (conclude that the data were not statistically significant enough to be believed to be nonrandom)	Reject the null hypothesis test (conclude that the data were statistically significant enough to be believed to be nonrandom)
Data is random	The right outcome	Type I error
Data is not random	Type II error	The right outcome

There is, however, a significant problem concerning this table, and it is that a randomness test can never meet the requirements of the statistical hypothesis test shown in the table. No matter how

good a randomness test is, it is always possible to construct a nonrandom sequence which will meet its criteria.

Imagine a perfect randomness test which can be run on a Turing machine. In order for it to be completely certain that a sequence is not random, the sequence needs to be infinite in length. The perfect randomness test cannot accomplish this, since traversing an infinite sequence would take infinite time on a Turing machine. In other words, the program would never halt. A Turing machine cannot complete such a task. Therefore, the ideal randomness test cannot be certain that a sequence is not random.

Consider the perfect randomness test. For any sequence of finite length as the input, the perfect randomness test will give as output a Boolean value. If the output value is true, it means that the null hypothesis was rejected and if it is false, the null hypothesis was accepted. Depending on what sigma used in the test, after having passed a sequence of some length N, the perfect randomness test will be able to return the output 'true'. Imagine a random number generator which goes through all sequences of length N (note that this is still a finite number of computations), evaluates them with the perfect randomness test and outputs the first sequence that passes the perfect randomness test. In other words, the first sequence of length N, which when evaluated by the perfect randomness test, results in the output 'false'. Such a random number generator is definitely not random, as all it uses are completely deterministic algorithms that can be run on a Turing machine. Therefore, we have a random number generator which, when evaluated by the perfect randomness test, will always result in the Type II error. In short:

For any sequence of finite length N, a nonrandom sequence can be generated using a Turing machine which will not result in the null hypothesis being rejected when evaluated by the best randomness test that can be run on a Turing machine.

Thus, the table for a randomness test is actually the following.

	Accept the null hypothesis test (conclude that the data were not statistically significant enough to be believed not to meet the criteria)	Reject the null hypothesis test (conclude that the data were statistically significant enough to be believed not to meet the criteria)
Data should match the criteria	The right outcome	Type I error
Data should not match the criteria	Type II error	The right outcome

If the input were truly random, it should always match the criteria, no matter what randomness test is used.

2.2 Sigma and p-values

Sigma, most often written as lowercase σ, and p-value are both used for rejecting the null hypothesis of a statistical hypothesis test (see chapter 2.1). In a statistical hypothesis test, the objective is to gather enough evidence in order to reject the null hypothesis. The data can be converted into a single value: the p-value or sigma. Then, by comparing the value to the predetermined p-value or sigma cutoff point (common ones are p-value less than or equal to 0.05 and sigma more than or equal to 2), we can determine whether to reject the null hypothesis or not.

P-value stands for probability value. The smaller the p-value the lower the probability of the outcome. The p-value is a real value between 0 and 1. A p-value of 0.01 means that if the null hypothesis is true, the chance of an outcome equally or more unlikely as the one obtained is 1%. A common p-value used as the cutoff point for rejecting the null hypothesis is 0.05. Below is a simple example of how to use the p-value in a statistical hypothesis test.

We have a coin which we think is biased in such a way that when we toss it, the chance of heads is greater than tails. We toss the coin three times. The outcome is two heads and one tails. There are a total of eight different combinations: HHH, HHT, HTH, HTT, THH, THT, TTH and TTT. Our result was HHT. There are four different throws where the number of heads matches or is more than our result: HHH, HHT, HTH and THH. Thus, our p-value is 4/8 = 0.5, meaning that if the coin is fair, there is a 50% probability of two heads or more. Note that this was a one-tailed test where only larger than the excepted number of heads would have been able to reject the null hypothesis. The null hypothesis was that the number of heads is less than or equal to half of the throws and, therefore, even if the coin only landed on tails, the null hypothesis could not have been rejected. The coin landing only on tails would actually have resulted in the p-value 8/8 = 1, since zero heads would have been the complete opposite compared to the statistical hypothesis test in question.

If we were to change the null hypothesis to heads and tails being equally likely, our p-value would have been one. This is because two heads and one tail or one head and two tails is the closest that three tosses can reach a uniform distribution. This two-tailed test requires a result that deviates from the mean more than the one-tailed test since both high and low values are tested for, instead of only the high or only the low value.

If our result were HHH instead of HHT, the one-tailed test would result in a p-value of 0.125. Out of all eight possible outcomes, only HHH has more or the same number of heads as HHH. If we throw HHH during the two-tailed test, there would exist two outcomes that match HHH in |number of heads – expected number of heads| (absolute value of the subtraction). These are HHH with the value 1.5 and TTT with the value 1.5. This means that our p-value would be 2/8 =

0.25. In both cases, the p-value was above 0.05, meaning that the null hypotheses could not be rejected. It was actually impossible to reject the null hypotheses for all sequences of length three, because three tosses is too few to result in p-values below or equal to 0.05, no matter what the sequences were. When testing random number generators, it is not uncommon to use millions of random bits in order to be able to reject the null hypotheses.

As experiments are often much more complicated than the simple coin toss bias experiments explained earlier, the use of sigma instead of the p-value has typically been favored, both because it is easier to use compared to the p-value when using very low probabilities and because it indicates that the data of the experiment was normally distributed. Sigma stands for the standard deviation used in a normal distribution. The data obtained from most experiments follow the normal distribution, making it intuitive to use sigma to explain how unlikely the results are if they were solely created by random fluctuation. Sigma can also be converted into a p-value. However, in order to convert a p-value into sigma, we must first ensure that the data follow the normal distribution. Note that for sigma we also need to define if the test was one- or two-tailed, since a one-tailed sigma and a two-tailed sigma result in different p-values. The sigma value starts from 0 and can be any positive real number, although integers are mostly used when discussing the results of experiments. For example, when the Higgs boson was discovered, they claimed the certainty to be of 5 sigma in a one-tailed test [13], which corresponds to a p-value of $2.867*10^{-7}$. In order to show where sigma is useful, we need a more advanced example.

We test if a coin is biased by tossing it a hundred times and then count the number of times it lands on heads. Our null hypothesis is that the coin is fair. The expected number of heads is 50, as this is the mean when tossing a fair coin a hundred times. Since we only want to know if the coin is biased, the test will be two-tailed. A significantly low number of heads would reject the null hypothesis as well as an abnormally large number of heads. We choose two sigma as the cutoff point.

The number of heads when tossing a fair coin a hundred times follows the following pattern:

$$\sum_{k=0}^{100} \binom{100}{k} * 0.5^{k} * 0.5^{-k} = 1$$

where $\binom{100}{k}$ is the combination, defined by: $\binom{x}{y} = \frac{x!}{y!(x-y)!}$

In this sum, k represents the number of heads we received from tossing the coin one hundred times. By adding up all of the 101 possible results (0 to 100 heads) we understandably arrive in the probability 1. The calculation takes a long time to complete by hand and concerning larger experiments, it can even be difficult for a computer. Luckily, this type of calculation, the binomial distribution, when using large numbers, has an asymptotically precise approximation, the normal distribution.

Let us say that the results from the 100 coin tosses was 68 heads. Our cutoff point was two sigma in order to reject the null hypothesis. The mean number of heads from tossing a fair coin one hundred times is 0.5 * 100 = 50 and the standard deviation is $\sqrt{0.5 * 0.5 * 100}$ = 5. Throwing 68 heads is (68 – 50) / 5 = 3.6 standard deviations away from the mean. Since 3.6 sigma is greater than or equal to 2 sigma, we reject the null hypothesis. Note that the result 3.6 sigma was only obtained from this single test and that the resulting sigma will most likely differ in subsequent tests. Actually, if the coin is unbiased, the resulting sigma would be random, following the normal distribution. If we were to calculate the p-value from sigma, it should also be random, following the uniform continuous probability distribution between 0 and 1.

2.3 Inner workings of a PRNG

Most pseudorandom number generators have many properties in common. These properties will be utilized when random number generators are analyzed and compared in subsequent chapters, meaning that a good understanding of these properties is required to understand the rest of the thesis. Thus, the goal of this chapter is to explain the many common properties of RNGs.

In order for a pseudorandom number generator to generate a number it requires at least two properties: a state and a formula, which creates the output number from that state and updates the state. Once the PRNG has generated a new number, it will then update the state in order not to generate the exact same number again. Depending on the PRNG, the state may be the last generated number, a simple counter or something completely different. The number generated may even be created only from a part of the state. A simple example would be the Lehmer random number generator, which was mentioned in chapter 1. Its general formula is the following:

$S_{N+1} = A * S_N \bmod M$

where A is the multiplier used on the state S and M is the modulus. For example, if A is 5, M is 31 and S_N is 17, S_{N+1} could be calculated with (5*17) mod 31, which is 23. Thus, if the state was 17, the new state (and number generated) would for this Lehmer random number generator always be 23.

Pseudorandom number generators are algorithms, which are by definition deterministic. This should mean that the same PRNG will always generate the same output. However, this is not true, since most PRNGs use a seed. A seed is the initial state of a random number generator. Its purpose is to ensure that a PRNG does not generate the same values each time the PRNG is used for the first time. In the Lehmer PRNG example, the seed would be state zero, S_0. The seed is usually

created from something that appears partially random. Examples include current time, key timings and mouse movements or other human input. In operative systems using Linux, a kernel random number generator exists, which places “environmental noise from device drivers and other sources into an entropy pool” [14], which can then be used for creating seeds or even for generating random numbers. A common method is to use current time, and then use a hash function on it to create the seed.

A defining feature of a PRNG is how much space it uses. Space is measured in bits and consists mostly of the PRNG’s state. In the case of the Lehmer random number generator, the state is limited by modulus M, meaning that in our example we would need five bits to store the state. PRNGs have a specific number of bits of randomness they generate in one pass. Most PRNGs generate either 32 of 64 bits in one pass. The example Lehmer random number generator would generate $\log_2(31) \approx 4.95$ bits of randomness in one pass (if M would have been 32, it would be exactly 5).

Another defining feature of a PRNG is its period length. The period length is the number of values it can create before the PRNG repeats itself. The period length is closely related to how much space the PRNG uses, since an upper limit on the period length of a PRNG can be calculated from how much space it uses. As the PRNG will always generate the same values from a specific state, the number of states it can reach is the same as its period length. In our example, the state is limited by M, which was 31, meaning that the state could only be an integer between 0 and 30, which, if it were able to traverse all those integers, would result in a period length of 31. We could also use its space usage to calculate the maximum possible period length, which in this case would be $2^5 = 32$. PRNGs try to use their space as efficiently as possible, trying to reach this largest possible period length. Note that it is possible to create a PRNG which does not have a period length, but such a PRNG would either require more true randomness (as in an ever-changing seed) or increasing amounts of space. PRNGs without period lengths tend to be slow compared to those with a fixed period length and, thus, they are used for more specialized tasks, such as cryptography.

2.4 Defining a good PRNG

Now that we have defined randomness and some methods for measuring it, the next step is to define what a good pseudorandom number generator entails. Any number generator can be seen as a pseudorandom number generator, although potentially a really inferior one. Before trying to create a decent definition, we need to specify what the use case is for the random number generators, as this

can greatly affect the definition of ‘good’.

There are two types of pseudorandom number generators: those that are cryptographically secure and those that are not. The cryptographically secure PRNGs are usually slower and require more true randomness than just a single seed. The cryptographically secure random number generators (CSPRNG) have a special condition which other random number generators do not need to fulfill. A CSPRNG must generate a sequence which cannot be used to calculate its current state, given that one knows how the random number generator works. In other words, it should not be possible to predict the outcome of a CSPRNG only by examining its output. This means that no PRNG that uses its state as the generated number can ever be a CSPRNG. As this thesis has so far handled PRNGs in general, we will also create the definition for PRNGs instead of only CSPRNGs. Therefore, not being able to calculate the current state from the sequence generated is not one of the requirements used in our definition of a good pseudorandom number generator. The optimal PRNG should be good enough for simulations and computer games, that is, the non-gambling type games.

Thus, a good PRNG needs to fulfill four requirements: it needs to be fast, use as little space as possible, have a long period length and generate pseudorandomness of good quality. Note that pseudorandomness of good quality is a very vague description. Note also that pseudorandomness of good quality requires a long period length, but a long period length does not imply pseudorandomness of good quality. Long period lengths cannot be tested by brute-force, meaning that one needs to prove that a PRNG really has the claimed period length. In order to ensure that a PRNG generates pseudorandomness of good quality, a battery of randomness tests is required to ensure that it passes at least some sort of check. Randomness tests will be discussed in detail in chapter 3.

3 Randomness Tests

A randomness test can be seen as a function. The input comprises a stream of bits of a predetermined length and the output is a single real value between zero and one. The input is the random sequence that we want to test and the output is the p-value. The resulting p-value should follow the continuous uniform distribution if the given random sequence truly is random. Therefore, the p-value tells how likely the outcome was; the smaller the p-value the less likely the outcome. If the p-value is extremely small, it could mean that the given random sequence is not random at all.

Given that the actual test can be anything, it is important to try to define what a useful random number test would be. A random number test which takes as input only one bit and checks if it is a one is far from useful. If the random bit which we test is a one, we would obtain the p-value 0.5 and if it is a zero the p-value would be one, meaning that we cannot even achieve the standard 0.05 p-value in any of the two possible test outcomes. Note that this is a one-tailed test. If we were to make the test into a two-tailed test, where we check how far from a half the given bit is, the p-value would be one for both zero and one. One can argue that there cannot exist a more useless random number test than one which gives the p-value one for all inputs. One can also argue that a random number test which cannot reach a low enough p-value is useless. Although it is difficult to define ‘low enough’, it is still always possible to define it to be equal to or less than the p-value required to reject the null hypothesis in question. This, however, is difficult to use since the p-value required to reject the null hypothesis is arbitrarily chosen and thus varies considerably. Therefore, useful randomness tests might be completely useless in other experiments. A much better solution would be to say that given any p-value x you can always define a finite sequence of bits, which would, when submitted to the random number test, result in a p-value less than x. I call this the requirement of any desired p-value.

This is, of course, far from what is needed for a random number test to be useful. Consider the following random number test. The test checks if the given random bits are all equal to one. If we were to give the test ten bits, for 1023 of all of the 1024 combinations of bits, it would give the p-value one and for only one of the combinations, the one consisting of only ones, it would give the p-value 1/1024, a little less than 0.001. A random number test that is too specific, like the given example, is also useless since it will only be able to give a p-value low enough in very few situations. For any given p-value x, as the length of the random numbers being tested tends to infinity, the fraction of all combinations of that specific length which result in a p-value below x should not tend to zero. I call this the requirement of nonzero p-values.

Using these two constraints as the base requirements for a random number test is not enough for the random number test to be useful, but it is a good start. The remaining requirements that I

have pondered on are much more difficult to define mathematically, if not impossible, and some are practically completely dependent on which random number generator you want to test. The remaining requirements are defined in chapter 3.1.

One of the simplest useful random number tests is the Monobit test. This is the same test as the one used in chapter 2.2 to explain how p-value and sigma work. In the Monobit test, we measure the occurrence of ones and zeros compared to what would be the excepted value from a true random sequence of the same length. The two-tailed test is deemed to be much better, since we can test for both extremes, although we will most often need more bits to test in order to obtain the p-value we want in the cases where we would be able to reach the desired p-value. It is rather trivial to prove that this random number test satisfies both the requirement of any desired p-value and the requirement of nonzero p-value. Given a sequence of length k, the sequence containing only ones, the p-value would be $2/2^k$ (if the test were one-tailed, the p-value would be $1/2^k$) which can be smaller than any given value since k can be arbitrarily large and, thus, the requirement of any desired p-value is fulfilled. In order to prove that the test fulfills the requirement of nonzero p-value we will use sigma. Since the Monobit test, given enough bits, tends towards the normal distribution, it is trivial to choose any sigma and with the help of it calculate both the p-value and the two ratios of ones and zeros which will give that p-value (since the test is two-tailed).

The Monobit test is a very simple example of a random number test, yet still very useful. Several tests have been created over the years, including the diehard tests and the tests found in the NIST Statistical Test Suite [17]. When testing a random number generator, multiple tests are used. Multiple tests are used because different tests test different aspects of non-randomness of a random number generator. It is actually more useful to have multiple smaller tests than one larger test, since it makes it much easier to pinpoint where the random number generator contains flaws and clues might even exist concerning how to correct these flaws.

Many of the randomness tests are rather complicated. A single test could have been an effort of multiple people working together for months or even years. This is mainly because the mathematics in randomness tests becomes quite complicated rather quickly. It is often required to integrate functions which do not have any analytical function to express the integral of the function. This makes it rather difficult to calculate precise values, even with the use of a computer, requiring multiple optimizations and approximations in order to even come close to a correct p-value. In order to tackle this problem, the software I have made will mostly be using Pearson's chi-square test to calculate p-values. The idea is to transform any interesting randomness tests into a variant of Pearson's chi-square test. The chi-square test is explained in detail in chapter 3.2.

3.1 Defining a good Randomness Test

In chapter 3, two important qualities concerning randomness tests were explained with examples. These were “requirement of any desired p-value” and “requirement of nonzero p-value”. These two requirements are, however, far from enough to determine what a good randomness test is. The remaining methods I have devised are “requirement of minimizing value usage”, “requirement of more is more”, “requirement of not being too specific”, “requirement of not being too complex” and “requirement of not being perfect”.

The requirement of minimizing value usage means that a randomness test should never require a large number of generated random values in order to calculate the p-value. For example, a Monobit test which checks every tenth bit in a sequence would throw away nine tenths of all generated bits, which seems wasteful, even if this type of Monobit test might detect lack of randomness in cases where the Monobit test which uses all bits fails to do so. Since we do not know how fast a PRNG that we test is, a good randomness test should waste as few bits as possible.

The requirement of more is more is tied to the requirements of any desired p-value and nonzero p-value. The idea is that for a PRNG which behaves in a manner which leads to a randomness test T to conclude that it is not random after being ran X times, should be able to give a more extreme p-value after having run for more than X times, since it failed when it was run X times. The reason why this requirement cannot be mathematically defined is that it simply will not work for all PRNGs when it comes to large X values. A PRNG that uses the Champernowne constant to generate its pseudorandom numbers will at first fail any frequency test (see chapter 4.3.1) but since it ultimately is a normal number, it will pass the tests given enough runs. Thus, this requirement can be applied only when the PRNG tested has a finite period length.

The requirement of not being too specific has to do with creating a test that tests something very specific. A randomness test specifically designed for some type of pseudorandom number generators, such as the spectral test, is most likely not a good randomness test outside of testing that type of pseudorandom number generator. Another example would be to use the exact same algorithm the PRNG uses to test it and to use the same initial seed. This would result in a perfect match for that specific PRNG but most likely no matches concerning other PRNGs, and as such, it is much too specific.

The requirement of not being too complex has to do with algorithm runtime. We might be able to create a very good randomness test which runs in O(N!) time complexity, where N is the number of bits tested by the randomness test. Due to the complexity of the test, it will always be far from feasible. Randomness tests should never go below O(N), as then they fail to follow the

requirement of minimizing value usage, nor should they go above polynomial time. Some new interesting randomness tests may be created using quantum computers, where something that would have a non-polynomial runtime on a classical computer, would have a polynomial runtime on a quantum computer. Some of the best randomness tests have a big-O notation above linear, such as the ranks test which, when calculated with Gaussian elimination, has a running time complexity of $O(N^3)$.

The requirement of not being perfect is a more abstract one, since a perfect randomness test cannot be created, as the Kolmogorov complexity test is incomputable. The idea of this requirement is that one should not even try to create the perfect randomness test, as it defeats the point of testing pseudorandom number generators. If the goal is to try to measure the quality of randomness in a PRNG, what is the use of a randomness test which only tells us that the PRNG was not random with 100% accuracy? This places randomness tests in an odd light, since perfection is not desirable, which sounds slightly too philosophical for being a mathematical concept, or better yet, a statistical concept.

3.2 Pearson's chi-square test

Pearson's chi-square test [15], which we will call the chi-square test from now on, is a test in which the randomness that is to be tested is separated into a predetermined finite number of categories, and then compared to the theoretical mean values of these categories in order to calculate a p-value. The chi-square test can also be used to compare two or more samples, but we are only interested in comparing one sample to the theoretical categorical distribution it should have. The chi-square test is an approximation in the same sense as the normal distribution is an approximation of the binomial distribution. The more bits that are tested, the better the approximation is. This means that p-values calculated using the chi-square test differ from the true values, but this difference is minimal when using a large number of bits, since the relation between the p-value obtained from the chi-square test and the true p-value is asymptotic.

The chi-square test is more than a test. It is a tool that can be used to calculate p-values for almost all kinds of randomness tests. As long as we test for discrete distributions of finite size, it should be possible to turn it into a chi-square test. A simple example would once again be the Monobit test. We have two categories, one containing the zeros and the other containing the ones. The theoretical categorical distribution is 50% for each of the two categories. The chi-square test is always a two-tailed test and, thus, it will always detect all types of extremes in a distribution.

In the way the chi-square test was defined earlier, it is possible to have any finite number of categories. Imagine a version of the Monobit test where we have four categories: 00, 01, 10 and 11. This could be called the Twobit test. The categorical distribution is 25% for each of the four categories. In this case, the chi-square test is a two-tailed test in three dimensions. This is because knowing the total number of bits tested and the number of values in three of the four categories, it is trivial to calculate the number of values in the fourth category. Thus, there is a dependency. If there are N categories, there exists N-1 independent values. The concept is often referred to as the test following the chi-square distribution with X degrees of freedom, where X is the number of independent values. The chi-square distribution is a family of distributions where the X degrees of freedom dictate what the distribution looks like. The chi-square distribution is used by the chi-square test to calculate the p-values. The Monobit test follows the chi-square distribution with one degree of freedom and the Twobit test follows the chi-square distribution with three degrees of freedom. It is useful to look at the degrees of freedom as dimensions in which the test is two-tailed. Given some p-value as the cutoff range to reject the null hypothesis, it could be seen as a three-dimensional object in the case of the Twobit test. If the values in the four categories lead to a value outside the aforementioned three-dimensional object, the null hypothesis is rejected.

Note that the categorical distribution must be discrete but does not have to be uniform. A test where we count the number of ones in two bits would have three different categories, a category containing zero ones consisting of 00, a category containing a single one consisting of 01 and 10 and a category containing two ones consisting of 11. Thus, the probabilities are 25% for zero ones, 50% for a single one and 25% for two ones. This is a valid chi-square test where the categorical distribution is not uniform. This test follows the chi-square distribution with two degrees of freedom.

Since the idea of the chi-square test is to be an X-dimensional two-tailed test, we can apply the law of large numbers. Say that we have a true random number generator which generates ones 51% of the time and zeros 49% of the time, we could, given enough bits, obtain a p-value as close to zero as we want by using the Monobit chi-square test. This means that the Monobit chi-square test can be used to detect differences in the probability distribution of a true random number generator. Even if the random number generator is a true random number generator, as long as its distribution is not precisely what we compare it to, (most often the uniform discrete distribution with zero and one) we could reject the null hypothesis. If the Monobit chi-square test can fail a random number generator, it can be argued that the Twobit chi-square test can also fail the same random number generator. This, however, only applies to true random number generators, as only they have true probability distributions, as a probability distribution technically does not exist for something that is pseudorandom.

If we have the same true random number generator which generates bits with the distribution 51% probability for ones and 49% probability for zeros, it will be possible to detect the difference compared to the uniform discrete distribution with the Twobit chi-square test. We have four categories: 00, 01, 10 and 11. In order for the test to detect non-randomness we need to guarantee that the probability for at least one of the four categories is different from the expected probability. We will actually guarantee that at least two values are different, since changing the probability of one category will change the probability of at least one of the other categories. The law of large numbers will handle the rest. Therefore, the objective is to distribute 51% ones and 49% zeros so that all four categories have the correct probabilities. This can be proven to be impossible since 25% of 00 gives 25% zeros, 25% of 01 gives 12.5% zeros and 12.5% ones, 25% of 10 gives 12.5% ones and 12.5% zeros and 25% of 11 gives 25% ones. Adding up 25% + 12.5% + 12.5% gives us exactly 50% zeros, and 12.5% + 12.5% + 25% gives exactly 50% ones, meaning that there is no way of disguising 51% ones and 49% zeroes so that the distribution would not change. This is true also for the test that counts ones in two bits.

However, the opposite of this does not work. If we had a case where the Twobit chi-square test rejects the null hypothesis because all of the bits have landed in the 01 category, the Monobit chi-square test would not detect any non-randomness since both zero and one would still have the correct distribution. An N-bit chi-square test should detect non-randomness that all M-bit chi-square tests can detect, where $M < N$. This is true as long as the number of bits is large enough, and the random number generator is a true random number generator. Things become rather complicated if we are not testing a true random number generator as a probability distribution is required, which as earlier explained, is something that a PRNG lacks.

The main drawback for the chi-square test is that in situations, where we have a several degrees of freedom or the probability for one or more of the categories in the distribution is really low, plenty of data is needed to ensure a valid p-value. As the chi-square test is only an approximation, the use of many categories and categories with low probabilities will make the approximation less accurate, which means that more data was required to achieve a decent approximation. This, however, is not an issue, since randomness tests which use a multitude of categories in their distributions are often breaking the “requirement of not being too specific”-rule. Interestingly, any randomness test created with the chi-square test follows at least three of the rules explained in chapters 3 and 3.1. These rules are “requirement of any desired p-value”, “requirement of nonzero p-values” and “requirement of more is more”.

3.3 XOR Scrambling

The main objective of analyzing randomness tests is to determine if the test can detect lack of randomness in RNGs of high quality. In order to provide more RNGs to test, it would be useful to be able to combine generators to create new RNGs. This is the main idea of XOR scrambling.

Imagine two random number generators R_1 and R_2 and two randomness tests T_1 and T_2. Presume that R_1 passes the test T_1 but not the test T_2 and R_2 passes the test T_2 but not the test T_1. Could we somehow combine the output from R_1 and R_2 in order to create a random number generator R_3, which would pass both tests? One option would be to take the output from R_1 and R_2 and calculate the bitwise XOR (exclusive or). Below is the truth table for XOR:

R_1	R_2	$R_3 = R_1$ xor R_2
1	1	0
1	0	1
0	1	1
0	0	0

As we can see from the table, as long as one of the two random number generators generates true randomness, the XOR will also be truly random. What if both R_1 and R_2 do not generate true randomness? Is there anything we could still say about XOR scrambling them?

There are a couple of things that can be deduced analytically. XOR will always try to even out the number of zeros and ones if one of R_1 and R_2 generates true randomness. The true randomness does not need to be uniform. This is proven below:

If R_1 generates the bit 0 with a probability of X and the bit 1 with the probability of 1-X and R_2 generates the bit 0 with a probability of Y and the bit 1 with the probability of 1-Y, we can combine them in order to obtain the probabilities of R_3. We use the table from earlier and add the probabilities:

R_1	R_2	$R_3 = R_1$ xor R_2
1: (1-X)	1: (1-Y)	0: (1-X)(1-Y)
1: (1-X)	0: Y	1: (1-X)Y
0: X	1: (1-Y)	1: X(1-Y)
0: X	0: Y	0: XY

The probability of R_3 generating a zero is $(1 – X)(1 – Y) + XY = 2XY – X – Y + 1$

The probability of R_3 generating a one is $(1 - X)Y + X(1 - Y) = -2XY + X + Y$

What we now want to prove is the following statement:

The probability distribution of R_3 is closer or equally close to the discrete uniform distribution compared to both R_1 and R_2.

If the probability of R_3 generating one as an output is closer or equally close to half of the time when compared to both R_1 and R_2 the statement would be proven true, as there are only two discrete values.

This can be written as the following equations:

equation 1: $| -2XY + X + Y - 0.5 | \leq | 1 - X - 0.5 |$

equation 2: $| -2XY + X + Y - 0.5 | \leq | 1 - Y - 0.5 |$

where $0 \leq X \leq 1$ and $0 \leq Y \leq 1$

We only need to prove one of the two equations since equation 1 is equal to equation 2, where X and Y have swapped places. Therefore, we only prove equation 1:

$| -2XY + X + Y - 0.5 | \leq | 1 - X - 0.5 |$ simplify

$| -2XY + X + Y - 0.5 | \leq | 0.5 - X |$

Now we need to split the equation into two parts, one where $X \leq 0.5$ and one where $X > 0.5$

Here we prove the statement to be true when $X > 0.5$

$\| -2XY + X + Y - 0.5 \| \leq \| 0.5 - X \|$	$X > 0.5$
$\| -2XY + X + Y - 0.5 \| \leq X - 0.5$	rewrite
$-X + 0.5 \leq -2XY + X + Y - 0.5 \leq X - 0.5$	add $X - 0.5$
$0 \leq -2XY + 2X + Y - 1 \leq 2X - 1$	

Next, we look at the case $0 \leq -2XY + 2X + Y - 1$

$0 \leq -2XY + 2X + Y - 1$	rewrite
$0 \leq Y(-2X + 1) + 2X - 1$	rewrite $2X - 1$ to become Z, which is a value that can vary between [0, 1]
$0 \leq -YZ + Z$	rewrite
$0 \leq Z(1 - Y)$	

Since Z is within [0, 1] and $1 - Y$ cannot be less than zero, we conclude that $0 \leq Z(1 - Y)$ is true.

Next, we prove $-2XY + 2X + Y - 1 \leq 2X - 1$

$-2XY + Y \leq 0$ add $-2X + 1$

$Y(1 - 2X) \leq 0$

Since Y is a value in [0, 1] and (1 – 2X) < 0 is true, since X > 0.5, the result will always be negative or equal to zero, proving the second half of the case where X > 0.5.

Next, we prove the statement when $X \leq 0.5$

$| -2XY + X + Y – 0.5 | \leq | 0.5 – X |$ $X \leq 0.5$

$| -2XY + X + Y – 0.5 | \leq 0.5 – X$ rewrite

$-0.5 + X \leq -2XY + X + Y – 0.5 \leq 0.5 – X$ add 0.5 – X

$0 \leq -2XY + Y \leq 1 – 2X$

Next, we look at the case $0 \leq -2XY + Y$

$0 \leq -2XY + Y$ rewrite

$0 \leq Y(1 - 2X)$

Since Y is a value in [0, 1] and 1 – 2X is always larger or equal to zero, since $X \leq 0.5$, we can conclude that the result will always be positive or equal to zero.

Finally, we prove $-2XY + Y \leq 1 – 2X$

$-2XY + Y \leq 1 – 2X$ add -1 + 2X

$-2XY + Y + 2X – 1 \leq 0$ rewrite

$Y(-2X + 1) + 2X – 1 \leq 0$ rewrite 2X – 1 to Z, a value in [-1, 0]

$-YZ + Z \leq 0$ rewrite

$Z(1 – Y) \leq 0$

Since Z is in [-1, 0] and 1 – Y is in [0, 1], Z(1 – Y) must be smaller or equal to zero.

Thus, $| -2XY + X + Y – 0.5 | \leq | 1 – X – 0.5 |$ is proven to be true.

This would mean that at least for the Monobit test, combining two random number generators should improve the results. If XOR scrambling helps in more complicated tests is unknown.

Another thing which can be deduced analytically is that as long as one of the two generators outputs true randomness, the resulting combined RNG will also generate true randomness. The proof is much simpler than the previous proof.

Assume that R_1 is a true random number generator and R_2 generates pseudorandom numbers. What we want to prove is that the combination $R_3 = R_1$ XOR R_2 will remain random.

By looking at the first table in this chapter we can see that if R_2 generates a zero, there are two outcomes depending on what R_1 chooses: 50% probability of R_3 being zero and 50% probability of being one. This can also be seen if R_2 generates a one. Combining these two possible outcomes we can see that no matter what R_2 generates, R_3 will remain random.

The interesting part about the XOR scrambling is that no matter the outcome, the results will still be very useful. If XOR scrambling would actually worsen the quality of pseudorandomness, that is to say make the new random number generator fail more tests, this information could perhaps be used to create better randomness tests. The two PRNGs that were XOR scrambled should then have some sort of hidden property in common. If combining two pseudorandom number generators in this manner could create a generator, which passes at least the union of the set of tests which each of the generators passed separately, will be tested.

4 The Software, Specification

In order to study randomness tests, a program is developed in Java. The goal of the program is to provide an interface in which the user can create tests by choosing specific random number generators and randomness tests. The user can also combine multiple random number generators and randomness tests in order to study the inner workings of said generators and tests. The random number generators and randomness tests will provide methods for changing parameters which will change their behavior. Allowing the combination and changing of generators and tests provides the user with a very flexible tool for testing almost any aspect of a randomness test. If the inclusion of parameters and the use of multiple tests is not enough, the user can also create his own randomness tests from scratch using the interfaces that can be found in the source code. This should give the user practically limitless possibilities for testing randomness tests.

The program allows full support for using multiple threads, speeding up tests of larger scale. Single tests may not be sped up due to the fact that most random number generators cannot be parallelized reliably. Thus, the speedup of using multiple threads is limited by how many different random number generators are being tested. For example, if we give the program eight threads, but we are only running tests for five random number generators, five of those eight threads will be in use. Since some tests require more random numbers, and random number generators vary in execution speed, the tests may be split into smaller chunks so that all threads will finish their execution in approximately the same time.

In some cases, larger tests may be needed. Tests may take from only seconds up to hours or even days to finish. When a test may take several hours to complete, regular backups of the data are needed. The user should be able to pause the tests and resume at a later point or save finished tests and load tests to look at the results at a later point. In order to allow all of these functions, the random number generators need to have a state that one can go back to. Java has an interface called ‘serializable’. With the help of this interface the exact state of any object, including random number generators and randomness tests, can be saved and loaded. Almost all classes found in the Java standard library implement serializable, meaning that the random number generators found in the standard library can have their state saved and loaded. Other random number generators may be found in third party libraries or they need to be implemented from scratch, meaning that as long as one remembers to have the random number generators implement serializable, loading and saving should not be a problem.

Since most random number generators in use pass all but the most complicated tests, some of the random number generators that can be tested have been deliberately made to fail in even the simplest of tests. These deliberately bad random number generators can also be used for trying to

find out what sort of lack of randomness different randomness tests can detect.

4.1 The Software, Implementation

The name of the program is RATT, which stands for RAndomness Test Tester. Below is a picture of the main page of RATT. It contains three sections, one for the generators, and another for the tests and a third one for executing tests and for viewing test results in real time.

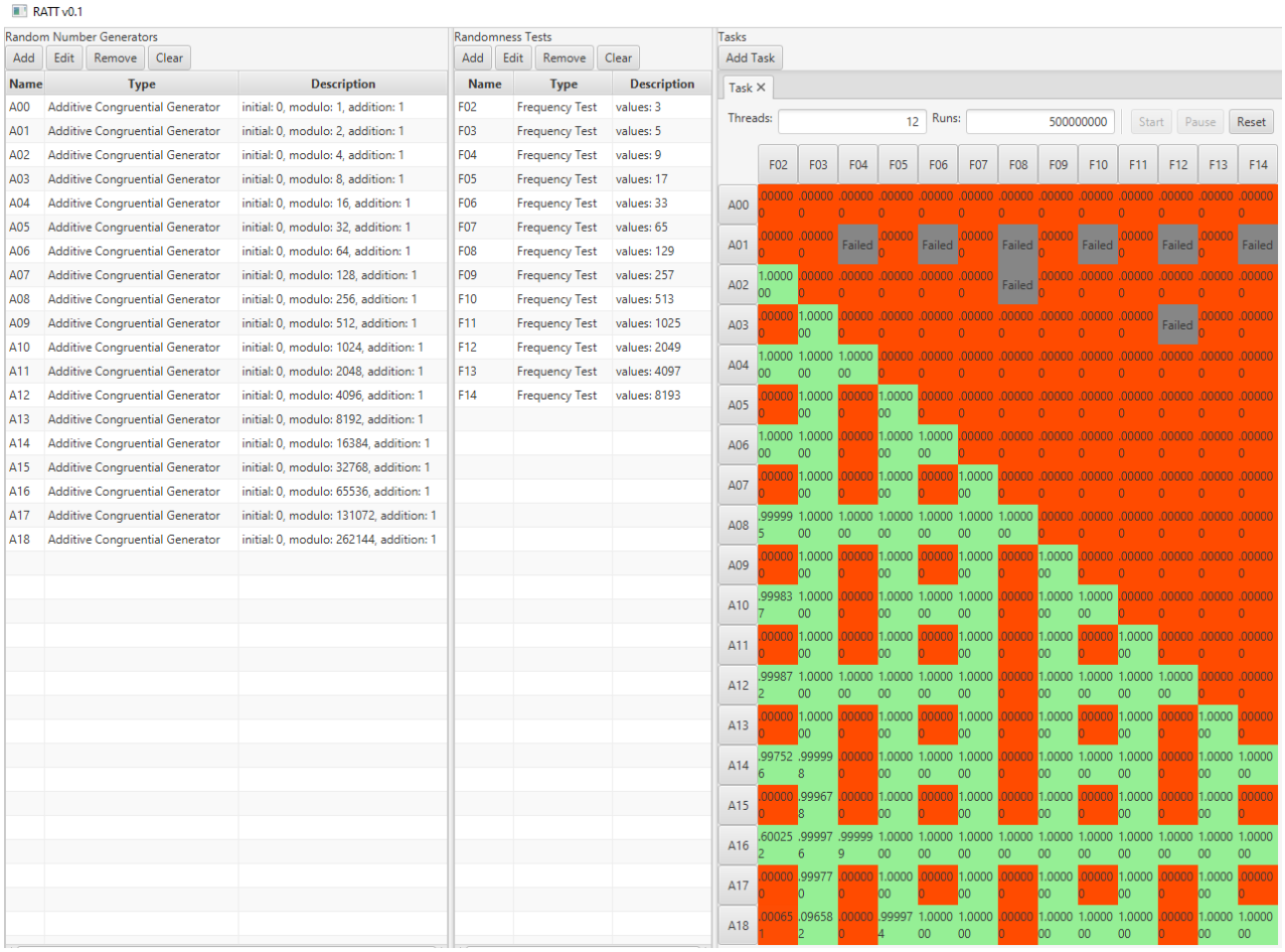

Figure 1. The main page of RATT version 0.1

RATT uses a library called apache commons-math3 in order to calculate the chi-square tests. The library contains a method where two arrays of values are given as input, one containing the expected distribution in percentiles, and the other containing the observed data. The library uses these two arrays to calculate the p-value.

4.2 The Random Number Generators

In order to test how well randomness tests work, a multitude of different random number generators are required. This includes both good generators and ones that create random numbers of poor quality. Thus, RATT contains both good and intentionally bad generators.

When explaining the different random number generators a couple of abbreviations will be used. PL stands for Period Length, gcd for greatest common divisor and mod for modulus. Any abbreviations not mentioned here will be explained when used.

Since PL is an important part of a PRNG, attempts will be made to calculate and prove different random number generator PLs.

4.2.1 Congruential Generator

The congruential generator (CG) is a random number generator that follows the following recurrence relation:

$$X_{n+1} = (f(X_n)) \bmod M$$

where X_n is the current state of the generator and, thus, X_0 is the seed given to the generator. In the RATT implementation, X_0 needs to be greater or equal to zero. M is the modulus. The function f(x) defines the congruential generator. The Linear Congruential Generator (LCG) is a subset of the congruential generator where the recurrence relation is of the type:

$$X_{n+1} = (A * X_n + C) \bmod M$$

This means that the function f(x) for LCG is $f(x) = A * x + C$

The congruential generator defined in RATT allows for the following recurrence relation:

$$X_{n+1} = (S * X_n^2 + A * X_n + C) \bmod M$$

The values that can be changed are S, A, C and M. In addition, one can define which bit range should be the output since the least significant bits can often provide randomness of lower quality.

Depending on how the congruential generator is configured, the period length can be difficult to calculate. The highest possible period length is M. In the special case that S is 0 and A is 1, the exact period length can be calculated with the simple formula:

$$PL = M/\gcd(C,M)$$

This is the case when the recurrence relation $X_{n+1} = (X_n + C) \bmod M$.

Proof:
Since we can have at most M states, and the states are deterministic, we can say that if we have two states X_m and X_n where m does not equal n, the period length is at most abs(m-n). Thus, in order to calculate the period length we need to minimize abs(m-n).

When calculating $(X_n + C) \bmod M$ the result can be one of two values, $X_n + C$ or $X_n + C – M$. Therefore, the result could be written as $X_n + C – b * M$, where b is some integer larger or equal to zero. This procedure can be continued, for example, the next value would be $X_n + 2 * C – b * M$. The value that C is multiplied by tells us how many iterations we have done since X_n. If we have a situation where some $X_n + a * C – b * M = X_n$, it means that a is a multiple of the period length. In order to calculate the period length, we need to find the smallest possible a.

Thus, the equation to solve will look like this:

$$X_0 = X_0 + a * C – b * M$$

where all values are integers, $a > 0$ and a is minimized.
As we can see, the X_0 cancel out, meaning that our starting location will not matter concerning the period length. The equation can therefore be simplified into the following Diophantine equation:

$$a * C – b * M = 0$$

which can be further simplified to:

$$C/M = b/a$$

One solution would be b = C and a = M, but could a be smaller than M? In order to find the smallest possible a, we simply need to reduce C/M into their lowest terms, which is calculated with the use of the greatest common divisor, thus, a = M/gcd(C,M) and b = C/gcd(C,M). This means that if and only if C and M are coprime (as in gcd(C,M) = 1), we have the period length of M.

In case S is not equal to 0 or A is not equal to 1, calculating the period length becomes more difficult. Things become even more complicated since, depending on the seed used, the period length may vary. A simple linear congruential generator defined below has this property:

$$X_{n+1} = (2 * X_n + 2) \bmod 7$$

If $X_0 = 0$, the period length is three: 0, 2, 6, 0, 2, 6...
If $X_0 = 1$, the period length is three: 1, 4, 3, 1, 4, 3...
If $X_0 = 5$, the period length is one: 5, 5, 5, 5, 5, 5…

The Dull-Dobell Theorem [16] explains when a linear congruential generator has the highest possible period M if C is not equal to 0. The theorem states that PL = M if and only if:

(1) M and C are coprime (same as the requirement when A = 0)

(2) 1 = (A) mod P, for any P that is a prime factor of M

(3) 1 = (A) mod 4, if 4 is a factor of M

This means that if M is a power of two, the only requirements are that C and A should not be divisible by two in order to reach the maximum period length M.

The congruential generator should be useful for testing randomness tests, since it can take the form of multiple random number generators that have been used in actual programs. This includes all linear congruential generators, quadratic congruential generators, Lehmer random number generators and even a version of the binary Champernowne sequence.

4.2.2 Runs Generator

The runs generator is a simple generator meant to generate a stream of a specific length consisting only of either ones or zeros. The stream is reset by appending one bit of the opposite value. For example, if we were to generate streams of length three consisting of ones, the output of the runs generator would be:

11101110111…

As a generator which will immediately fail the Monobit test ([17] 2-2), it will be useful to determine whether a randomness test might be a superset of the Monobit test or not.

4.2.3 Java Random Number Generators

There are a couple of random number generators in the Java standard library, meaning that no programming language translation needs to be done. The random number generators that have been added to RATT are java.util.Random and java.util.SplittableRandom. A class exists called java.security.SecureRandom which I would have wanted to add to RATT, but could not. Random and SecureRandom are both serializable but SplittableRandom is not. This problem can be easily solved, since the source code of SplittableRandom can be copied into its own SerializableSplittableRandom class, and then have it implement serializability. The state of the SplittableRandom consists of two values of type long, meaning that serializability should not be a problem.

Java.util.Random is a linear congruential generator that could be created with the congruential

generator explained in chapter 4.2.1, but it is added to the list of random number generators, since it is found in the Java standard library and, therefore, is most likely being frequently used. The class uses the recurrence relation:

$$X_{n+1} = (A * X_n + C) \,\&\, B$$

where B is 2^{48}-1, A is 645 534 673 and C is 11 and & is the bitwise AND-operator. The bitwise AND-operator is used to calculate mod B+1 much faster, since 2^{48}-1 consists of 47 ones when represented in binary. Thus, the recurrence relation is equal to the LCG:

$$X_{n+1} = (645\,534\,673 * X_n + 11) \bmod 2^{48}$$

Since C is not equal to zero, we can check if the random number generator has the maximum period length of 2^{48} with the Dull-Dobell Theorem, which was explained in chapter 4.2.1. The first requirement is that M and C should be coprime, which is the case since $\gcd(11,2^{48}) = 1$. The second requirement is that A mod P should be one for any P being a prime factor in M. The only prime factor of 2^{48} is 2. A mod 2 is 1, meaning that the second requirement is fulfilled. The third requirement is that if four is a factor of M, A mod 4 should be 1, which is also true. Therefore, the period length of java.util.Random is 2^{48}. The output of the generator leaves out the 16 least significant bits, meaning that the total number of bits the generator can generate before the sequence repeats itself is $2^{48} * 32$. This number of bits should be enough for some applications, as long as the generated randomness is not of poor quality. It would take approximately 300 hours for a single threaded program run on an AMD Ryzen 7 3700X to generate all of the $2^{48} * 32$ bits.

Java.util.SplittableRandom uses two values of type long to store its state, the seed and gamma. The internal state of SplittableRandom works like a congruential generator with 2^{64} as the modulus:

$$Seed_{n+1} = (Seed_n + Gamma) \bmod 2^{64}$$

Then, the seed is mixed by a function to generate the output. The algorithms for mixing the seed are different for 32 and 64 bit outputs. Below are the two functions taken directly from the source code:

```
/**
 * Computes Stafford variant 13 of 64bit mix function.
 */
private static long mix64(long z) {
    z = (z ^ (z >>> 30)) * 0xbf58476d1ce4e5b9L;
    z = (z ^ (z >>> 27)) * 0x94d049bb133111ebL;
    return z ^ (z >>> 31);
}

/**
 * Returns the 32 high bits of Stafford variant 4 mix64 function as int.
 */
private static int mix32(long z) {
    z = (z ^ (z >>> 33)) * 0x62a9d9ed799705f5L;
    return (int)(((z ^ (z >>> 28)) * 0xcb24d0a5c88c35b3L) >>> 32);
}
```

Here, the ^ stands for the bitwise XOR operator and >>> for the zero fill right shift operator.

The period length of SplittableRandom can be determined with the use of the proof in 4.2.1, which states that the period length of $X_{n+1} = (X_n + C) \bmod M$ is M/gcd(C,M). In the case of SplittableRandom, M is 2^{64} and C is by default the value 10 437 801 985 508 215. Since M is a power of two and C is odd, gcd(C,M) must be one. Thus, the period length is M, which is 2^{64}. This is, however, only true for the internal 'counter' of the SplittableRandom class. The documentation of SplittableRandom claims that the period length is at least 2^{64} but a proof is not included. There seems to be no public documentation containing the proof. In order to ensure that the period length is 2^{64} the functions mix64 and mix32 need to be analyzed.

Since the state has the period length 2^{64} the only possible period lengths for SplittableRandom are ones that divide 2^{64}, which are the powers of 2 all the way from 2^0 up to 2^{64}. This means that if we can prove that the period length cannot be any of 2^0 up to 2^{63}, the period length must be 2^{64}. Since the internal state follows $X_{n+1} = (X_n + C) \bmod M$, it is trivial to calculate the state X_n, if we start with some value X_0:

$X_n = (X_0 + n*C) \bmod M$

As was stated earlier, the X_0 is of no importance in this calculation, thus, we can set $X_0 = 0$. In order to test mix64 and mix32 for periodicity, we need to first calculate the values mix64(0) and mix32(0). Interestingly both of them are zero, which means that the following statement must hold in order for the periodicity to be of length 2^{64}:

If we for all 2^m, where m is an integer and $0 <= m < 64$, prove that

mix64($(2^m * C) \bmod 2^{64}$) does not equal zero.

If this can be proven, it means that the period length for SplittableRandom, using the algorithm mix64 and the gamma value C, is 2^{64}. Note that this is not an 'if and only if' statement. Even if mix64($(2^m * C) \bmod 2^{64}$) for some value m equals zero, it still might not mean that the period length is not 2^{64}.

If we use the constructor for SplittableRandom, where we choose the seed, it will use the gamma value 10 437 801 985 508 215. Below is the code that was created in order to prove that a given gamma value ensures a periodicity of 2^{64}:

```
public static void testForFullPeriodicity(long gamma) {
	boolean mix32failed = false;
	boolean mix64failed = false;
	for (int i = 0; i < 64; i++) {
		BigInteger value = BigInteger.valueOf(gamma)
				.multiply(BigInteger.TWO.pow(i));
		if (mix32(value.longValue()) == 0) {
			mix32failed = true;
```

```
                }
                if (mix64(value.longValue()) == 0) {
                        mix64failed = true;
                }
        }
        System.out.println("Mix64 failed: "+mix64failed
                        +", mix32 failed: "+mix32failed);
}
```

The result, when running the program using the value 10 437 801 985 508 215, is:

```
Mix64 failed: false, mix32 failed: false
```

Note that SplittableRandom allows only specific values for gamma. It uses a function to ensure that the gamma value is odd (in order to reach the full period length of 2^{64}) and that it contains enough 0-1 or 1-0 bit transitions (for example 0011 would contain only one transition). Therefore, small gamma values, for example one, cannot be used. Even if the default SplittableRandom class does not allow the use of these small values, they might be of interest when testing randomness tests. Therefore, the user can define any gamma value for the splittable random generator in RATT.

The SecureRandom class is different from the other generators, as it is meant to be a cryptographically secure pseudorandom number generator (CSPRNG). Therefore, not that much information can be found of the actual implementation of SecureRandom, let alone any information about period lengths and, thus, it is not added to RATT.

4.2.4 Xorshift Generators

The xorshift generator was discovered by George Marsaglia to be used as a fast random number generator with randomness of good quality [18]. The initial random number generators only used two bitwise operators: the XOR-operator and bitshifting. This is why the name of the generator is xorshift. The xorshift family of random number generators have been shown to be statistically unreliable. This has led to new types of xorshift generators, which have introduced for example multiplication to the set of operators [19]. These have, however, also been proven to fail some randomness tests, often due to their least significant bits being of poorer quality of randomness. Note that the java class SplittableRandom is a xorshift generator with some changes made to it.

RATT contains some of the xorshift random number generators that Marsaglia mentioned in his paper “Xorshift RNGs” [18].

4.2.5 Permuted Congruential Generator

The permuted congruential generator (PCG) is a random number generator algorithm developed in 2014 by Melissa O'Neill [20]. PCG is not one single algorithm, but a family of algorithms, which are defined by a group of transformations. These transformations include multiplications of constants and multiple types of xorshift operations. The defining feature of a PCG is that its state is handled by an LCG, and the output is created by transforming this state with one or several transformations. The generated number is therefore not the state of the algorithm, which makes calculating the state more difficult, but not impossible. RATT contains the algorithm PCG-XSH-RR, which is one of the PCGs defined by O'Neill [21]. The algorithm has a 64-bit state and a 32-bit output.

Since PCG relies on an LCG, the PL of a PCG is at best the same as the LCGs, but it depends on the transformations used on the state, which for a general PCG is a more complicated calculation, but can be trivially calculated for a specific PCG, using the same method that was used for SplittableRandom in chapter 4.2.3.

4.3 The Randomness Tests

In this chapter, all of the randomness tests that are created for the software will be explained. Several of the tests are defined in the publication "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" [17]. The tests from the aforementioned publication that have been included in RATT will have their specific pages mentioned in the following explanations. As calculating p-values of randomness tests is rather difficult, and since RATT only supports Pearson's chi-square test, the tests will differ somewhat from the tests mentioned in the paper. The changes were made only to make the tests use the chi-square test. Thus, by having failed known RNGs, they should prove useful in testing the XOR scrambling theory explained in chapter 3.3.

4.3.1 Frequency Test

The simplest test to implement is the frequency test, where one simply divides the given input into N containers, and then calculates the p-value. The simplest frequency test is the Monobit test ([17]

2-2), where the input is divided into two containers: one containing all zeros and the other containing all ones. As we want to test a discrete uniform distribution, our null hypothesis is that the two containers are filled in a way that matches the uniform distribution. If we were to distribute N random bits, approximately half of them should be zeros and the other half ones. The paper contains a method for calculating the p-value with the Pearson's chi-square test. Therefore, only small changes had to be made in the calculations so that they would work with the interface RATT provides.

An example of the test in use has been provided in the paper. Four zeros and six ones were used as input, and the test resulted in a p-value of 0.527089. When tested with RATT, using java.util.random to create bits and Frequency Test with two values, the same p-value could be obtained when java.util.random generated four zeros and six ones. With the expected array being [0.5, 0.5] and the observation being [4, 6], the ChiSquareTest class provided the exact same result as was described in the publication.

The publication explains only the Monobit version of the frequency test, but RATT allows the user to divide data into any number of containers. In cases where the number of containers does not match a power of two, the test uses only the required number of bits to calculate a value, and if the value goes above the container count, it discards the value. Here is an example.

If the Frequency test uses five values (containers), then, the five containers would be: 000, 001, 010, 011 and 100. See figure 2 for an image taken from RATT showing the probability distribution of this configuration. In the case that the received random bits is 110011000111010 and we are using five values, the following steps would take place:

(1) Remove three bits from the beginning of the sequence. In this case, we would remove 110 and the remainder would be 011000111010. Next, go to step 2. If there are no more bits to remove, go to step 3.

(2) If the removed bits match a container, increase the count in that container and return to step 1. If the removed bits did not match a container, discard them and return to step 1. If we discard the removed bits a specific number of times in a row, stop the test and throw an exception.

(3) Use the chi-square test on the container data to calculate the p-value.

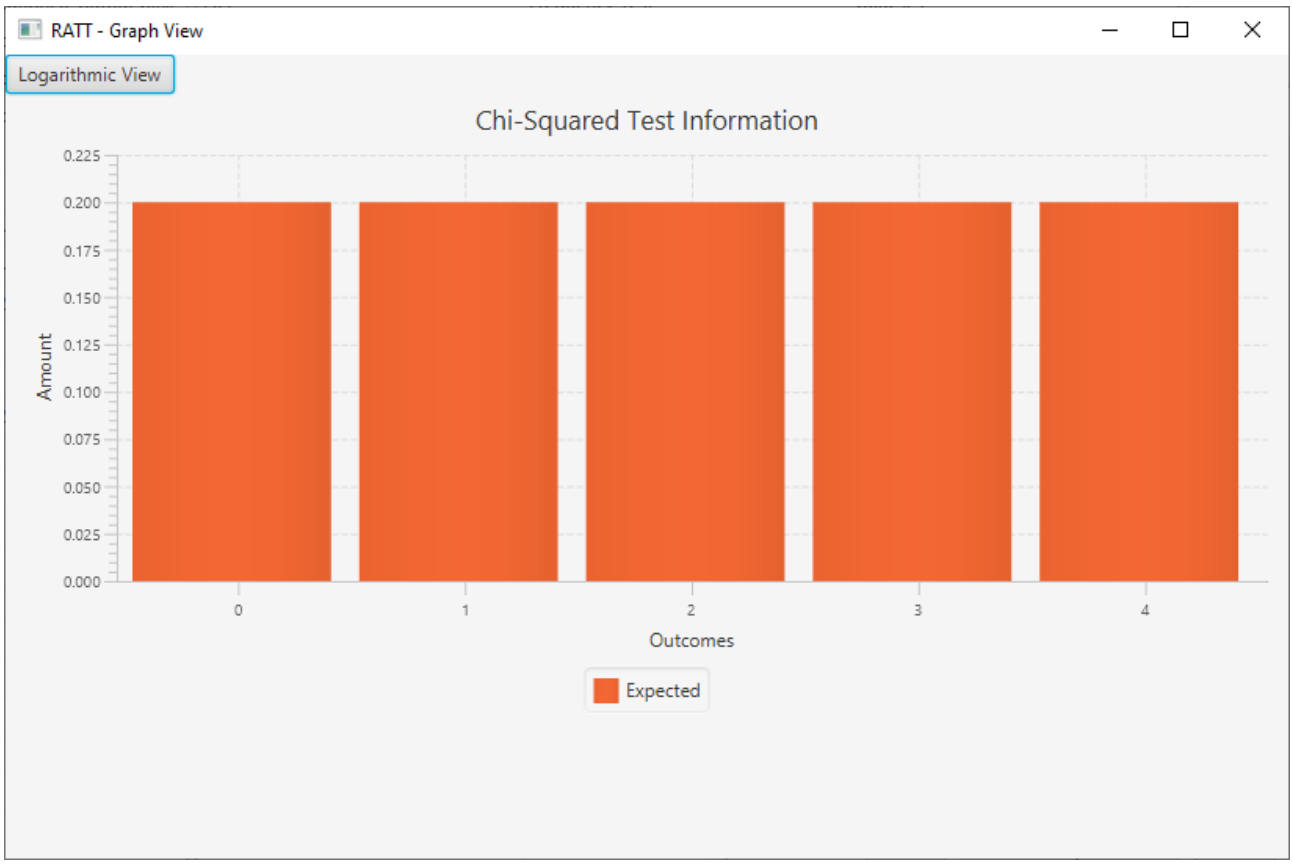

Figure 2. The statistical distribution of a Frequency test where the value count is five.

Albeit being a very simple test, the frequency test should prove useful as a rudimentary test which ensures that the implementations of random number generators, which surely should pass the frequency test, are correct.

4.3.2 Run Length Test

Similar to the Runs Test ([17] 2-3), the run length test measures the length of runs compared to calculating how many runs there are. A run is a sequence of bits of the same value. The run lengths are placed into N containers, where each run length, up to N, will be stored in a separate container. The Nth container stores all run lengths of length N and longer.

Calculating the expected statistical distributions is rather trivial, as the shortest possible run length is 1 and for each new bit and the probability of the run ending is always 50%. Thus, the probabilities will follow the simple formula below:

Run length of 1: 50%

Run length of N: (probability of run length of N-1)/2

Run length of anything longer than N: 100% minus the sum of all previous run lengths,

which is always the same as the run length of N.

Below is a simple example of the run length test. We use as input data the sequence 110011000111010, which was the same sequence used in the frequency test example. We will use three containers, for run lengths of one, two and three and above. There are three run lengths of two, two run lengths of three and two run lengths of one in the bit sequence. Note that the last bit in the sequence is not included, since we do not know how long its run length would have been. The expected array is [0.5, 0.25, 0.25] and the observed array was [2, 3, 2], resulting in a p-value of 0.455794.

The probability of a run of length N is subject to exponential decay, meaning that long runs are extremely rare. Therefore, the run length test should measure run lengths of at most 20, assuming that the number of random bits that can be provided is in the billions. Figure 3 demonstrates how the longer run lengths have a very small probability of occurring.

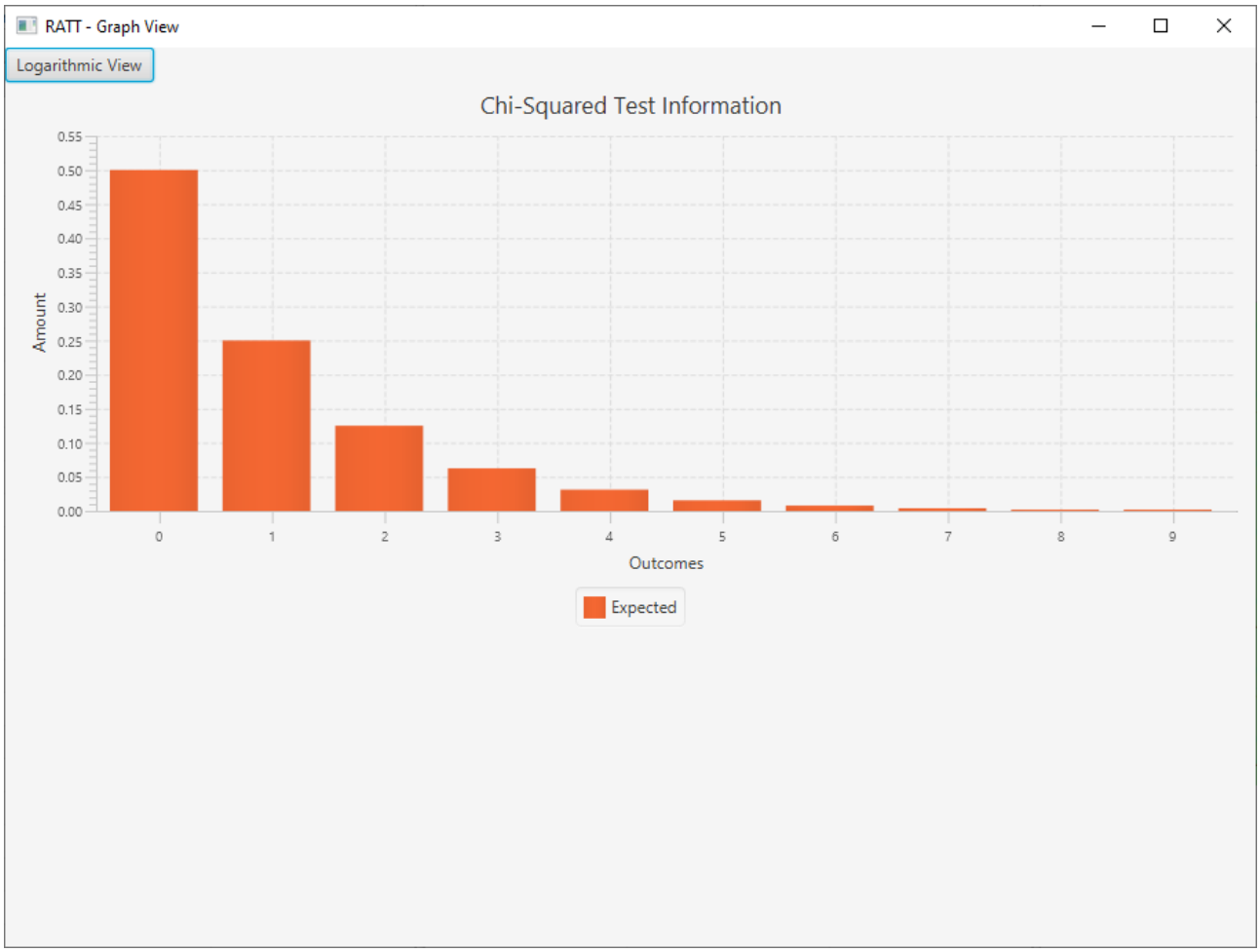

Figure 3. The statistical distribution of a run length test where the run lengths of up to 9 have their own container. The bar named 0 displays the probability of a run length of 1, bar 1 shows the probability of a run length of 2 and so forth.

4.3.3 Comparison Test

A test similar to the comparison test cannot be found in the publication “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”. The comparison test is somewhat similar to the frequency test 4.3.1. It selects values of some bit length S in the same way as the frequency test, but instead of selecting one value, it selects N of them. The test uses N containers. The N values are compared in size, and the largest value is placed into its container. If the largest value is shared by several of these N values, none of the values are placed into their containers. Below is an example.

If the bit length S is 2, the values range from 00 to 11. If N is 5, it would mean that we need to generate five values of size S, for example:

Value 1: 01

Value 2: 01

Value 3: 11

Value 4: 00

Value 5: 10

In this example, the third value was the largest, meaning that we increment the third container. Once we have generated enough data, we can compare the distribution found in the containers to the expected distribution and calculate the p-value.

The expected probability distribution is trivial to calculate. Since there is nothing that differs between the N values, all of them should have the same probability of being the largest value. There is no need to show a graph of the statistical distribution of the comparison test, since a comparison test, which uses N values, would have the exact same statistical distribution as a frequency test using N as its value. See figure 2 for an example of what the probability distribution for the comparison test would look like if N was five.

4.3.4 Ranks Test

The ranks test uses matrix ranks in order to create a randomness test. The idea is to create a binary matrix of some size using the Galois field 2, and then calculate its rank. Calculating the rank of an N by N matrix is a $O(N^3)$ operation which makes the ranks test the test with the highest complexity explained in this thesis. An algorithm exists that calculates the rank of a binary matrix, with Galois field 2, with the time complexity $O(N^2)$ [22]. There are a couple of specific ranks tests which are used more than others, namely the binary rank test for 31 by 31 matrices, binary rank test for 32 by

32 matrices and the binary rank test for 6 by 8 matrices. These three tests can be found in the diehard test suite, which was developed by George Marsaglia [23]. The probability distribution of the ranks test is approximative, since it is impractical to calculate the rank for every possible 32 by 32 and 31 by 31 binary matrix, as there are 2^{1024} respective 2^{961} of them.

I have implemented the 31 by 31 and 32 by 32 ranks tests in RATT, using the probability distributions found in the diehard source code. See figure 4 for the probability distribution of the 32 by 32 ranks test.

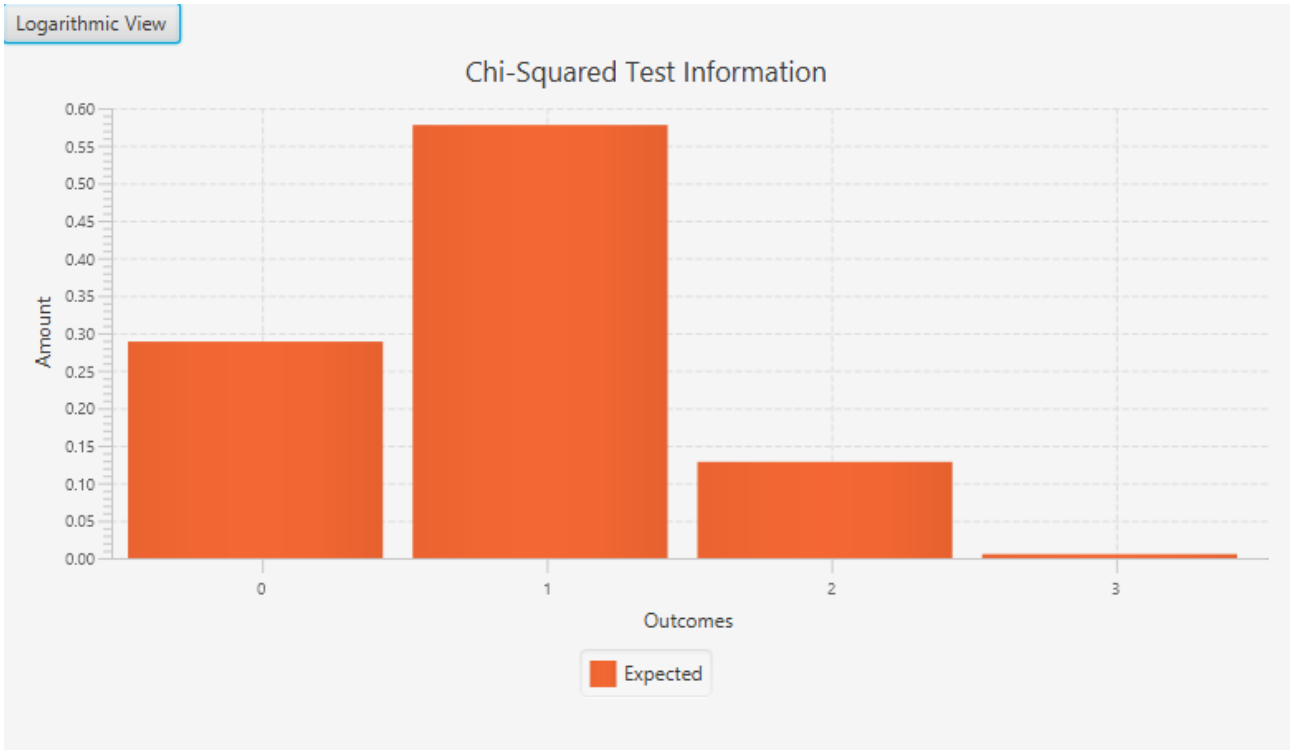

Figure 4. This is the probability distribution for the rank of a 32 by 32 binary matrix with the Galois field 2. 0 displays the probability of rank 32, 1 displays 31, 2 displays 30 and 3 displays 29 and lower.

4.3.5 One-Dimensional Random Walk Test

The one-dimensional random walk test is a simple yet potentially powerful randomness test. The bits used as input are used as movement commands. Draw the integer line, set the starting location to 0 and for each bit either move to the right of the current position by one if the bit was one, or move to the left of the current position by one if the bit was zero. Do this N times, store the final position and start from the beginning. This one-dimensional movement should produce a specific pattern, which can be verified with the chi-square test. The pattern in question is a version of Pascal's triangle. This can be deduced in the following way:

After step one, there is a 50% chance of being on -1 and a 50% chance of being on 1. Each

step doubles the number of walks that could be made. In order to avoid fractions we choose to multiply the probabilities by 2^N where N is the step count. Below is a table showing how the distribution unfolds as we increase the number of steps:

N	2^N	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	Sum
0	1							1							1
1	2						1		1						2
2	4					1		2		1					4
3	8				1		3		3		1				8
4	16			1		4		6		4		1			16
5	32		1		5		10		10		5		1		32
6	64	1		6		15		20		15		6		1	64

We can use this table to see what the probability is for each location depending on the number of steps taken. For example, after the sixth step has been taken, there is a 20/64 probability that we are at zero. Interestingly, after having taken an odd number of steps in total, one must be on an even integer, and after having taken an even number of steps in total, one must be on an odd integer. In order to make the distribution readable in RATT, the possible values that one can be in after N steps are indexed from 0 to up to N+1. For example, for four steps, -4 would be indexed as 0, -2 as 1, 0 as 3, 2 as 4 and 4 as 5. Figure 5 shows of the distribution for eight steps. The more steps that are taken, the closer to the normal distribution the distribution will be.

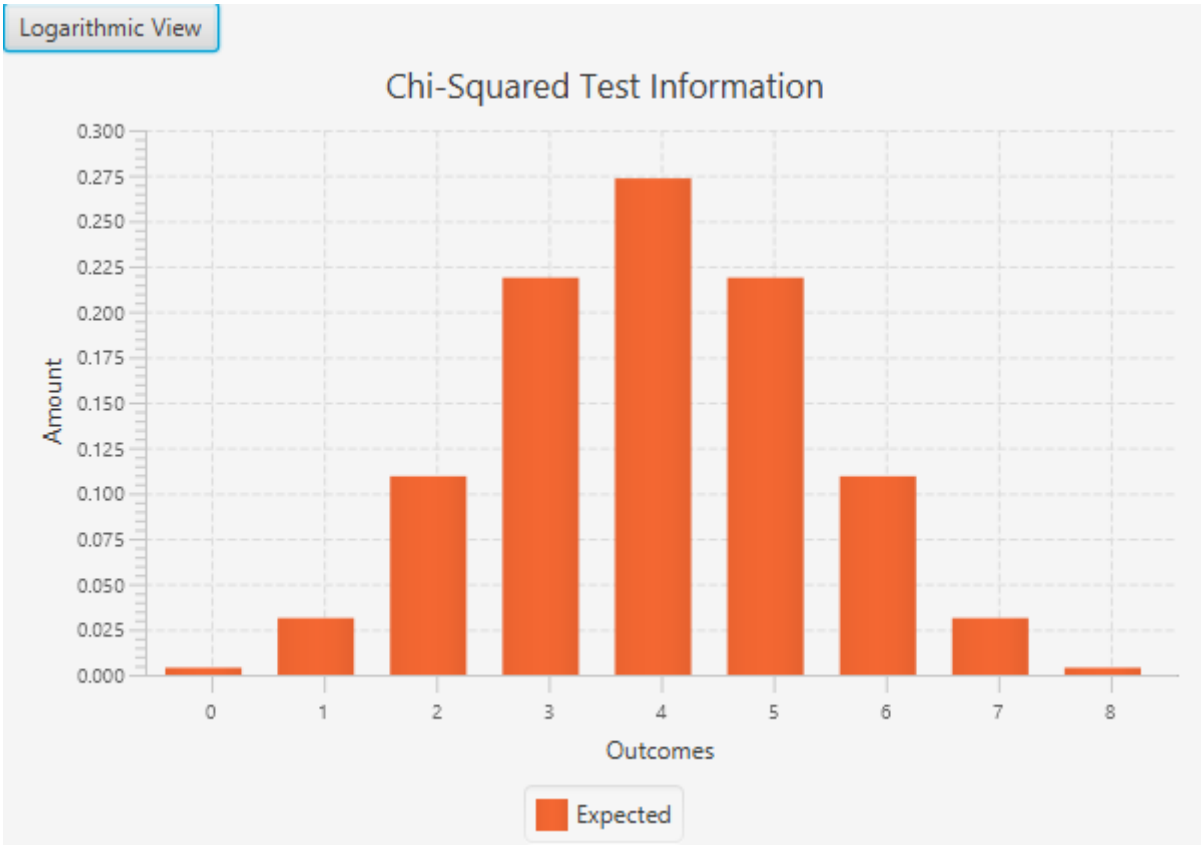

Figure 5. The probability distribution for a random walk of eight steps.

5 Results

Unsurprisingly, obtaining decent results from the tests proved to be difficult. The battle for superiority between pseudorandom number generators and randomness tests is tilted in favor of the generators as even the simple linear congruential generators pass most randomness tests provided its period length is large enough. The random number generators and the randomness tests that have been evaluated, as well as the results, can be seen in figure 6.

The outcome of this thesis could be seen in two different ways: either we have not yet found the holy grail of randomness tests or creating a PRNG that outputs random bits of high quality is surprisingly easy. If the latter is true, it seems PRNGs cannot be improved significantly more. For example, the 64-bit xorshift generator specified in George Marsaglia's paper "Xorshift RNGs" [18] does not require many binary operations to generate bits.

```
public long calculateNextValue() {
        state^=(state<<13);
        state^=(state>>>7);
        return state^=(state<<17);
}
```

A total of three bit shifts, three bitwise xor operations and three store operations on a 64-bit value is simple enough that it could be implemented in hardware instead of software. Other than that, there is not much more that can be improved concerning generation speeds. The period length will most likely never have to be larger than 2^{256} which is easily achieved with newer xorshift-based generators or permuted congruential generators (PCG) [20].

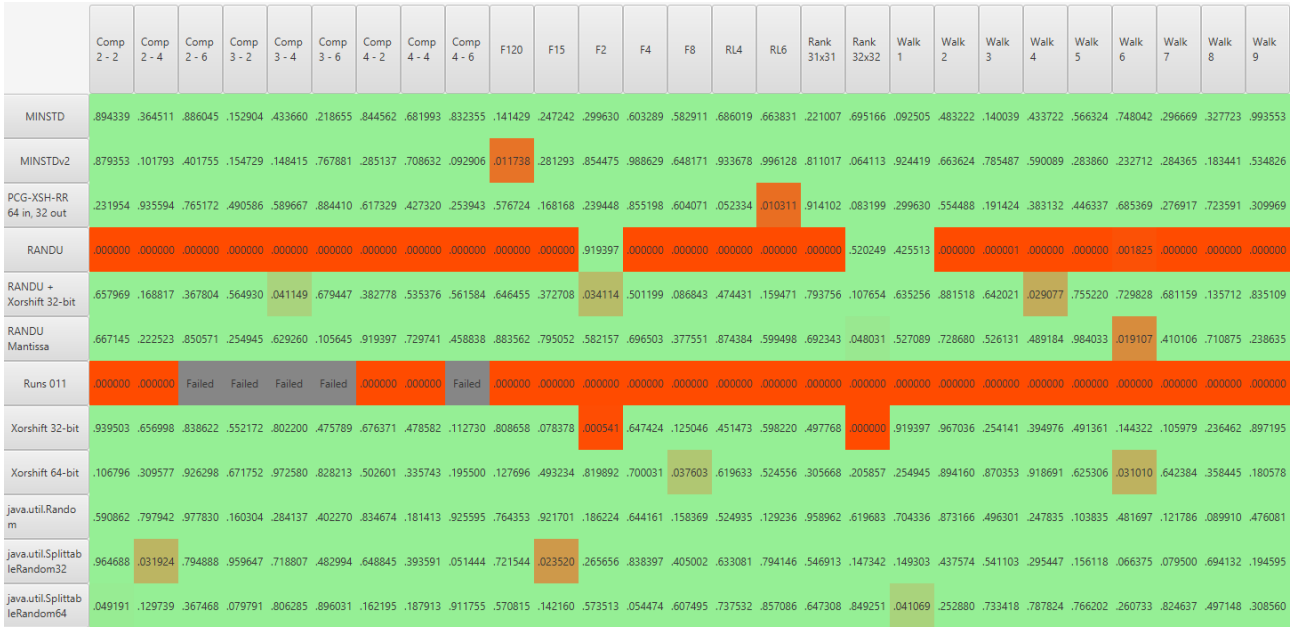

Figure 6. Each column is a randomness test. Comp X - Y is the comparison test where Y number of X bits are compared. F is the frequency test where the value after it denotes how many values are

used. Rank is the rank test. Walk is the one-dimensional random walk test. RL is the run length test. Each row is an RNG. MINSDT, MINSTDv2 and RANDU are specific Lehmer random number generators. RANDU mantissa discards the first eight bits that RANDU generates in one pass. RANDU + Xorshift 32-bit is an XOR combination of RANDU and Xorshift 32-bit. Xorshift and the java RNGs are self-evident.

As was explained earlier in this chapter, the random number generators seem to be winning the battle, when even the simplest random number generators can pass all of the randomness tests specified in figure 6. This made it difficult to test the XOR combination hypothesis but, luckily, xorshift 32-bit fails the 32x32 rank test and RANDU passes it, but fails at almost all other tests. The combination of xorshift 32-bit and RANDU managed to pass all tests, showing that at least for some PRNGs, an XOR scrambling of two generators will result in a better PRNG.

5.1 Patterns found in Linear Random Number Generators

Consider the frequency test for N values and the counting function:

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

where modulus M is some power of two. If N and M are equal, the result should never fail, but what about situations where they differ?

We define a counting function of modulus M to be C_M and a frequency test for N values to be F_N. What sort of pattern can be found if we have a group of counting functions, which have moduli of different powers of two, tested with a group of frequency tests with value counts also being different powers of two? Interestingly, the pattern is not trivial. Of course, we can quickly assume that for some F_N and C_M, F_N will pass if N and M are the same, and that F_N will fail if N is larger than M, but what about situations where N is smaller than M?

We need to look at this from a more analytical standpoint. We have C_8 tested with F_4. Since C_8 has a period length of only eight, we can easily check what the result is if we were to use an infinite number of bits on this test. The sequence of C_8 is:

000 001 010 011 100 101 110 111

Once split with F_4 the sequence will look like this:

00 00 01 01 00 11 10 01 01 11 01 11

which is a total of three 00, five 01, one 10 and three 11, yet the expected distribution should have been three of each of the four values. This means that for a C_M and F_N where N is less than M,

situations may exist where C_M fails the test.

This result is rather exciting since it means that, at least for some very specific random number generators, that is the counting functions, a larger frequency test does not encompass smaller frequency tests. The reason for this is, of course, divisibility.

For all C_M and F_N where $\log_2(N)$ divides $\log_2(M)$, C_M will pass the F_N test. This can be called the divisibility rule. We know that a full period of C_M contains exactly one of each value between 0 and M and that C_M is a superset for any C_K, where K is equal to or smaller than M and a power of two. This can be seen by removing leading bits in C_M. Below is an example of C_8 and C_4:

000 becomes 00

001 becomes 01

010 becomes 10

011 becomes 11

100 becomes 00

101 becomes 01

110 becomes 10

111 becomes 11

It does not even have to be the trailing bits, we could also use the first two bits:

000 becomes 00

001 becomes 00

010 becomes 01

011 becomes 01

100 becomes 10

101 becomes 10

110 becomes 11

111 becomes 11

Since $\log_2(8)$ cannot be divided by $\log_2(4)$ we cannot equally divide C_8, but if we had C_{16} this could be done and it would result in an equal distribution for all values, which means that C_{16} passes F_4. Therefore, using frequency tests with value counts of 2^N where N is a prime number could be recommended. This, interestingly, means that the worst possible frequency test is the Monobit test since all integers are divisible by one.

If C_8 cannot pass F_4, could we instead order the eight three bit values in some order which will pass F_4? Here is one solution:

000 001 010 011 100 110 101 111

00 00 01 01 00 11 10 01 10 10 11 11

Since eight tree-bit values exist, the number of combinations is 8! which is 40320. Out of these 40320, only 5952 manage to pass F_4. We call the PRNG which has a state of M bits, uses its state as the number output and has the period length of 2^M, which is the maximum possible for a state of M bits, P_M. The set of all P_M for some M is 2^M!, and the number of bits P_M creates during a full period is $M*2^M$. This means that in order to check by brute-force if some P_M passes a test F_N we will need to generate $M*2^M*\log_2(N)/\gcd(M*2^M, \log_2(N))$ bits, where gcd is the greatest common divisor. This means that for the set containing all P_M for some M, we would need to go through a total of $M*2^M*\log_2(N)/\gcd(M*2^M, \log_2(N))*2^M!$ bits, which for P_3, when tested with F_4, is already 967 680 bits. This makes calculating the percentage of all P_M for some M which pass F_N for some N extremely difficult. P_4 tested with F_8 would already require going through $4 * 10^{15}$ bits, or 446 tebibytes and already for P_5 brute-forcing would be impossible. Calculating P_3 tested with F_4 took around one second with the following python 3 program:

```
import itertools, re, math, collections
c_value = 8;f_value = 4
c_length = math.log(c_value, 2) * c_value
parts = c_length / math.log(f_value, 2) / f_value
multiplier = int(int(math.log(f_value,2))/math.gcd(int(math.log(c_value, 2)),
int(math.log(f_value,2))))
counter = 0; total = 0
for i in itertools.permutations(range(0,c_value,1)):
    current = ""; total += 1
    for j in i:
        current += format(j, '0'+str(int(math.log(c_value, 2)))+'b')
    tmp = current
    for j in range(1,multiplier):
        current += tmp
    d = dict(collections.Counter(re.findall('.'*int(math.log(f_value,2)),current)))
    if (all(v == (parts * multiplier) for v in d.values())):
        counter += 1
print(str(counter)+"/"+str(total))
```

If P_4 were to be tested with F_8 using this program, it would take approximately 50 years to complete. With the help of optimizations and a faster programming language, P_4 with F_8 should still be practical.

Even if calculating exact values is impossible, which leaves the whole idea more or less as a theoretical interest, the fact that C_M may pass F_N but fail at F_K even if $N = 2^A*K$ for some positive integer A and N and K are powers of two means that multiple frequency tests might be better than one single large test. More than that, it implies that one should use frequency tests of powers of J

bits, where J is a prime number. For example, if we were to test a large enough number of bits that frequency tests of up to F_{1000} are practical, we could then conclude that the only worthwhile frequency tests are F_{32}, F_{64}*, F_{128}, F_{256} and F_{512}, since F_2, F_4 and F_{16} are subsets of F_{256} and F_8 is a subset of F_{512}. Note that F_{64} has a star. This is because I am not sure if a combination of data provided from the tests F_{256} and F_{512} would encompass F_{64}. Why I think this could the case is that $64 = 2^6 = 2^{3*2}$, F_{256} contains 2^2 and F_{512} contains 2^3.

I created with RATT a table consisting of C_2, C_4, C_8…, C_{65536} and F_2, F_4, F_8…, F_{65536} and ran each test 47 233 105 920 times in order to ensure that each test is divisible by the bit length of the period of each random number generator (calculated with lcm($1*2^1$, $2*2^2$, $3*2^3$, …, $16*2^{16}$), where lcm is the least common multiple). This ensures that the p-values are precise. The calculation took 10 hours and 45 minutes using 14 threads on a R7 3700X processor. See figure 7 for the results of this calculation.

	F00002	F00004	F00008	F00016	F00032	F00064	F00128	F00256	F00512	F01024	F02048	F04096	F08192	F16384	F32768	F65536
C00002	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00004	1.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00008	1.000000	.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00016	1.000000	1.000000	1.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00032	1.000000	.000000	1.000000	.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00064	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00128	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00256	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C00512	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
C01024	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	.000000	.000000	.000000	.000000	.000000	.000000
C02048	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	.000000	.000000	.000000	.000000
C04096	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	1.000000	1.000000	.000000	.000000	.000000	.000000
C08192	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	.000000	.000000
C16384	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	1.000000	.000000	1.000000	1.000000	.000000	.000000
C32768	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000	1.000000	.000000
C65536	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Figure 7. Each row in the table represents a counting function of a specified length. Each column represents a frequency test of a specified length. Green means that the counting function on that row passed the frequency test on that column, and red means that it failed the test. The decimal values seen in the red and green areas are p-values.

It seems that for any C_M where M is $2^{(2^A)}$ and where A is an integer larger or equal to 0, C_M will pass all frequency tests F_N, where N is 2^B and smaller than or equal to M. I, however, do not know how to prove this. It also seems that a C_M, where M is 2^{2A+1}, will fail all tests F_N, where N is 2^{2B} and smaller or equal to M.

What makes this table interesting is that it follows the divisibility rule, yet in most cases where C is not divisible by F, it still passes the test. What the divisibility rule claims is that for

situations where C can be divided by F, it is not possible for C to fail F.

5.2 Unknown Probability Distributions

While creating these random number generators and studying how the calculations are made for them, I realized that creating analytical calculations for the probability distributions is both difficult and tedious. What makes this even worse is that the complexity of calculating the probability distribution is no guarantee of the randomness test being useful. In some cases, the probability distribution is too complicated to calculate analytically and, thus, only an approximation can be calculated (see the ranks test in chapter 4.3.4 as an example of one which is widely used). Therefore, when developing new randomness tests, the chi-square two sample test should be used before trying to calculate the probability distribution for the test.

Here are some simple rules of thumb when considering new randomness tests. Have one RNG which creates true randomness or pseudorandomness of the highest quality. Have it run the test along with a PRNG which does not need to be of high quality. Use the chi-square two sample test to determine if the two random number generators create differing distributions when tested with the new potential randomness test. The null hypothesis is that both random number generators create the same distribution, meaning that if the p-value is low, at least one of the random number generators would fail the test, if the test were to be properly implemented.

In this way, if most of the random number generators generate distributions similar enough that they do not fail the chi-square two sample test, we know that it would not be worth it to continue the development of that specific randomness test.

6 Further Research

As chapter 5 pointed out, random number generators are largely winning over their tests. Even the best of randomness tests can fail to detect simple congruential generators. More research should be focused on understanding why these really simple PRNGs manage to bypass tests with ease, as this would most likely also result in better randomness tests being discovered. According to the kolmogorov complexity test, the sequences the PRNGs generate are of inadequate quality due to how simple the algorithms are, yet according to all computable randomness tests made so far, the pseudorandomness they create is of a very high quality. The XOR combination hypothesis needs much more testing, since only one combination could be tested in this thesis.

The tool RATT works well, and multiple threads can be used to speed up larger tests. What the tool now needs is more customizability, more PRNGs to use for testing and more randomness tests.

7 Summary in Swedish – Mätandet av Slumpmässighet

Slupmässighet är ett svårförstått begrepp, främst för att den saknar en algoritm. En beräkningsbar algoritm klarar inte av att skapa annat än pseudoslumpmässighet. Då man jämför slumpmässighet med pseudoslumpmässighet är skillnaden teoretiskt sett mycket klar men blir praktiskt sett mycket svår, då enda sättet att urskilja dem är med slumpmässighetstester, som är mycket ineffektiva. Algoritmer som skapar pseudoslumpmässighet kallas för PRNG och står för "pseudorandom number generator", pseudoslumptalsgenerator.

Slumpmässighetstester använder sig av statistik för att säga hur sällsynt ett resultat är. Ifall resultatet är tillräckligt sällsynt för att nå en viss förutbestämd tröskel kan man säga att den PRNG som testades blev förkastad. Tröskeln för förkastandet mäts med hjälp av p-värdet. Ju mindre p-värdet är desto osannolikare är resultatet. P-värdet kan vara mellan 0 och 1. Om p-värdet är till exempel 0.05 betyder det att om slumpmässighetstestet har testat riktig slumpmässighet, skulle ett resultat som är lika extremt eller mer extremt som det vi fått hända med 5 % sannolikhet. Därmed kan mycket låga p-värden tyda på brister i en PRNG som testas.

Om målet är att skapa bra slumpmässighetstester måste man först definiera vad ett bra slumpmässighetstest innebär. Detta är inte en enkel uppgift då ett slumpmässighetstest inte får vara för bra. Vilken nytta har man av ett slumpmässighetstest som förkastar alla PRNG:n då de inte skapar riktig slumpmässighet? Därmed är ett slumpmässighetstest som använder sig av Kolmogorovkomplexitet [4.5 sida 9] för att beräkna ett p-värde inte vad vi söker efter (därutöver saknar ett sådant test en algoritm).

Av de krav som jag kom på är de tre viktigaste "kravet på önskat p-värde", "kravet på nollskilt p-värde" och "kravet på mera är mera". Kraven på önskat p-värde betyder att ett slumpmässighetstest, som ges något p-värde x, skall ha åtminstone en inmatning som får den att ge ut ett p-värde mindre än x. Kravet på nollskilt p-värde betyder att vid något p-värde x skall chansen att man får ett p-värde mindre än x inte bli noll då inmatningen växer mot oändligheten och inmatningen är slumpmässig. Med hjälp av dessa två krav kan man försäkra sig att ett slumpmässighetstest kan klara av vilket som helst för krav på p-värdet. Kravet på mera är mera är mycket likt de två tidigare kraven, men lägger till en mycket viktig sak. Om vi har ett slumpmässighetstest som förkastar en PRNG då vi använt N stycken värden, borde testet också förkasta samma PRNG om vi använder fler än N stycken värden, och då borde vi också få ett lägre p-värde. Då tredje kravet också uppfylls vet man att ju större antal värden man analyserar, desto lägre kommer p-värdet att bli, ifall den PRNG som testas kan förkastas av slumpmässighetstestet som används.

Jag har skapat programmet RATT (namnet härleds från "**RA**ndomness **T**est **T**ester") för att

studera slumpmässighetstester och algoritmer som skapar pseudoslumpmässighet. Med hjälp av RATT går det att jämföra olika PRNG:n sinsemellan och olika slumpmässighetstester sinsemellan. Jag har implementerat ett antal olika PRNG:n och slumpmässighetstester, jämfört dem och skrivit slutsatser om detta.

Alla slumpmässighetstester som RATT innehåller använder sig av Pearsons chi-två-test [15] för beräknandet av p-värdet vilket gör skapandet av fler tester enkelt, då de matematiskt krävande delarna alltid hanteras på samma sätt. Detta leder till vissa restriktioner gällande slumpmässighetstester men samtidigt uppfylls de tre viktigaste kraven som beskrevs tidigare.

Med RATT kan man också kombinera två stycken PRNG:n med hjälp av XOR-operationen. Om vi har två PRNG:n, R_1 och R_2, och två slumpmässighetstester, T_1 och T_2, för vilka R_1 klarar av T_1 men inte T_2 och R_2 klarar av T_2 men inte T_1 har jag följande hypotes; om man kombinerar R_1 och R_2 med hjälp av XOR-operationen kommer kombinationen att klara av både T_1 och T_2.

I figur 1 ses resultaten då jag jämfört alla PRNG:n med alla slumpmässighetstester. Varje kolumn är ett slumpmässighetstest och varje rad är en PRNG. Värdet som står i rutorna är p-värdet för de givna kombinationen av slumpmässighetstest och PRNG. I figur 1 ser man att då PRNG:n RANDU (klarade inte av värst många slumpmässighetstester) och Xorshift 32-bit (klarade av alla förutom Rank 32x32 testen) kombineras med hjälp av XOR-operationen, klarar denna kombination av alla test som den utsattes för. Detta betyder att åtminstone i vissa fall när man kombinerar PRNG:n med hjälp av XOR-operationen, kommer resultatet att skapa slumpmässighet av bättre kvalitet.

	Comp 2 - 2	Comp 2 - 4	Comp 2 - 6	Comp 3 - 2	Comp 3 - 4	Comp 3 - 6	Comp 4 - 2	Comp 4 - 4	Comp 4 - 6	F120	F15	F2	F4	F8	RL4	RL6	Rank 31x31	Rank 32x32	Walk 1	Walk 2	Walk 3	Walk 4	Walk 5	Walk 6	Walk 7	Walk 8	Walk 9
MINSTD	.894339	.364511	.886045	.152904	.433660	.218655	.844562	.681993	.832355	.141429	.247242	.299630	.603289	.582911	.686019	.663831	.221007	.695166	.092505	.483222	.140039	.433722	.566324	.748042	.296669	.327723	.993553
MINSTDv2	.879353	.101793	.401755	.154729	.148415	.767881	.285137	.708632	.092906	.011738	.281293	.854475	.988629	.648171	.933678	.996128	.811017	.064113	.924419	.663624	.785487	.590089	.283860	.232712	.284365	.183441	.534826
PCG-XSH-RR 64 in, 32 out	.231954	.935594	.765172	.490586	.589667	.884410	.617329	.427320	.253943	.576724	.168168	.239448	.855198	.604071	.052334	.010311	.914102	.083199	.299630	.554488	.191424	.383132	.446337	.685369	.276917	.723591	.309969
RANDU	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.919397	.000000	.000000	.000000	.000000	.000000	.520249	.425513	.000000	.000001	.000000	.000000	.001825	.000000	.000000	.000000
RANDU + Xorshift 32-bit	.657969	.168817	.367804	.564930	.041149	.679447	.382778	.535376	.561584	.646455	.372708	.034114	.501199	.086843	.474431	.159471	.793756	.107654	.635256	.881518	.642021	.029077	.755220	.729828	.681159	.135712	.835109
RANDU Mantissa	.667145	.222523	.850571	.254945	.629260	.105645	.919397	.729741	.458838	.883562	.795052	.582157	.696503	.377551	.874384	.599498	.692343	.048031	.527089	.728680	.526131	.489184	.984033	.019107	.410106	.710875	.238635
Runs 011	.000000	.000000	Failed	Failed	Failed	Failed	.000000	.000000	Failed	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000	.000000
Xorshift 32-bit	.939503	.656998	.838622	.552172	.802200	.475789	.676371	.478582	.112730	.808658	.078378	.000541	.647424	.125046	.451473	.598220	.497768	.000000	.919397	.967036	.254141	.394976	.491361	.144322	.105979	.236462	.897195
Xorshift 64-bit	.106796	.309577	.926298	.671752	.972580	.828213	.502601	.335743	.195500	.127696	.493234	.819892	.700031	.037603	.619633	.524556	.305668	.205857	.254945	.894160	.870353	.918691	.625306	.031010	.642384	.358445	.180578
java.util.Random	.590862	.797942	.977830	.160304	.284137	.402270	.834674	.181413	.925595	.764353	.921701	.186224	.644161	.158369	.524935	.129236	.958962	.619683	.704336	.873166	.496301	.247835	.103835	.481697	.121786	.089910	.476081
java.util.SplittableRandom32	.964688	.031924	.794888	.959647	.718807	.482994	.648845	.393591	.051444	.721544	.023520	.265656	.838397	.405002	.633081	.794146	.546913	.147342	.149303	.437574	.541103	.295447	.156118	.066375	.079500	.694132	.194595
java.util.SplittableRandom64	.049191	.129739	.367468	.079791	.806285	.896031	.162195	.187913	.911755	.570815	.142160	.573513	.054474	.607495	.737532	.857086	.647308	.849251	.041069	.252880	.733418	.787824	.766202	.260733	.824637	.497148	.308560

Figur 1. Varje rad är en slumptalsgenerator och varje kolumn ett slumpmässighetstest. Värden som syns är p-värdet då slumptalsgeneratorn i specifika raden blivit testad av slumpmässighetstesten i

specifika kolumnen.

Målet med RATT var inte att hitta brister i olika PRNG:n utan att jämföra slumpmässighetstester. Ifall ett slumpmässighetstest förkastar en delmängd av testade PRNG:n som alltid är en övermängd till de PRNG:n som ett annat slumpmässighetstest förkastar, finns det ingen nytta av att använda båda slumpmässighetstesterna. Då de flesta PRNG:n som testades klarade av antingen alla tester eller ett fåtal gick det inte att komma fram till någon slutsats angående slumpmässighetstester som skulle förkasta övermängder av vad andra slumpmässighetstester har förkastat.

Ur figur 1 framgår hur svårt det är att skapa bra slumpmässighetstester. Då PRNG:n oftast består av ett litet antal bitvisa operationer, additioner och multiplikationer, blir det svårt att hitta snabbare PRNG:n. PRNG:n som skulle skapa slumptal av bättre kvalitet är också en utmaning då slumpmässighetstester inte är tillräckligt utvecklade för att mäta nuvarande PRNG:n med hög standard. Det skulle löna sig att forska varför dessa enkla PRNG:n klarar av nästan alla slumpmässighetstester, då enligt Kolmogorovkomplexiteten är de fortfarande av låg kvalitet, men enligt beräkningsbara slumpmässighetstester skapar dessa PRNG:n slumpmässighet av hög kvalitet.

8 References

[1] Von Neumann *Various techniques used in connection with random digits, National Bureau of Standards Applied Mathematics Series, Volume 12,* pages 36-38, 1951

[2] D.H. Lehmer *Mathematical Methods in Large-Scale Computing Units, Symposium on Large-Scale Digital Calculating Machinery,* pages 141-146, 1951

[3] W.E. Thomson *A Modified Congruence Method of Generating Pseudo-random Numbers, The Computer Journal, Volume 1, Issue 2,* page 83, 1958

[4] Donald E. Knuth *The Art of Computer Programming – Seminumerical Algorithms, Third Edition,* page 93, 1997

[5] Manabendra Nath Bera et al. *Randomness in Quantum Mechanics: Philosophy, Physics and Technology*, *arXiv 1611.02176v2*, page 2, 2017

[6] A. N. Kolmogorov, *Foundations of the Theory of Probability*, page 2, 1950

[7] E. Wigner, *On the Quantum Correction for Thermodynamic Equilibrium, Department of Physics, Princeton University*, page 751, 1932

[8] Dirac, P. A. M. *Bakerian Lecture. The Physical Interpretation of Quantum Mechanics*, pages 1 and 7-8, 1942

[9] Gregory J. Chaitin, *Exploring Randomness*, page 113, 2001

[10] Cristian S. Calude et al, *Computing A Glimpse of Randomness,* page 1, 2002

[11] Andrei Khrennikov, *Randomness: quantum versus classical, arXiv:1512.08852v1*, 2015

[12] Gregory J. Chaitin, *A Theory of Program Size Formally Identical to Information Theory, Journal of the ACM 22*, page 19, 1975

[13] The CMS Collaboration, *Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC, arXiv:1207.7235v2*, pages 1 and 6, 2013

[14] man 4 random, can be found at *https://linux.die.net/man/4/random*

[15] Pearson, Karl, *On the Criterion that a given System of Deviations from the Probable in the Case of a Correlated System of Variables is such that it can be reasonably supposed to have arisen from Random Sampling*, *Philosophical Magazine*, pages 157-175, 1900.

[16] T.E. Hull; A.R. Dobell *Random Number Generators, SIAM Review, Vol. 4, No 3*, page 233, 1962

[17] Rukhin A. et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* Special Publication 800-22 Revision 1a, 2010

[18] George Marsaglia *Xorshift RNGs, Journal of Statistical Software*, 2003

[19] Daniel Lemire, Melissa E. O'Neill *Xorshift1024*, Xorshift1024+, Xorshift128+ and Xoroshiro128+ Fail Statistical Tests for Linearity, Computation and Applied Mathematics 350*,

2019
[20] Melissa E. O'Neill, *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation,* 2014
[21] Melissa E. O'Neill, *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation,* page 43*,* 2014
[22] Çetin K. Koç and Sarath N. Arachchige *A Fast Algorithm for Gaussian Elimination over GF(2) and Its Implementation on the GAPP*, Department of Electrical Engineering University of Houston*, page 118, 1991
[23] George Marsaglia, originally cited as http://stat.fsu.edu/pub/diehard/NOTES but now only found in https://web.archive.org/web/20160119080058/http://stat.fsu.edu/pub/diehard/NOTES, 1998