DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
ENGINEERING PHYSICS
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Analysis of Entropy Usage in Random Number Generators

JOEL GÄRTNER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ARCHITECTURE AND THE BUILT ENVIRONMENT

Analysis of Entropy Usage in Random Number Generators

JOEL GÄRTNER

Master in Computer Science
Date: July 2, 2017
Supervisor: Douglas Wikström
Examiner: Johan Håstad
Principal: Omegapoint
Swedish title: Analys av entropianvändning i slumptalsgeneratorer
School of Computer Science and Communication

Abstract

Cryptographically secure random number generators usually require an outside seed to be initialized. Other solutions instead use a continuous entropy stream to ensure that the internal state of the generator always remains unpredictable.

This thesis analyses four such generators with entropy inputs. Furthermore, different ways to estimate entropy is presented and a new method useful for the generator analysis is developed.

The developed entropy estimator performs well in tests and is used to analyse entropy gathered from the different generators. Furthermore, all the analysed generators exhibit some seemingly unintentional behaviour, but most should still be safe for use.

Sammanfattning

Kryptografiskt säkra slumptalsgeneratorer behöver ofta initialiseras med ett oförutsägbart frö. En annan lösning är att istället konstant ge slumptalsgeneratorer entropi. Detta gör det möjligt att garantera att det interna tillståndet i generatorn hålls oförutsägbart.

I den här rapporten analyseras fyra sådana generatorer som matas med entropi. Dessutom presenteras olika sätt att skatta entropi och en ny skattningsmetod utvecklas för att användas till analysen av generatorerna.

Den framtagna metoden för entropiskattning lyckas bra i tester och används för att analysera entropin i de olika generatorerna. Alla analyserade generatorer uppvisar beteenden som inte verkar optimala för generatorns funktionalitet. De flesta av de analyserade generatorerna verkar dock oftast säkra att använda.

Contents

Contents iii

Glossary vi

1 Introduction 1
- 1.1 Background 1
- 1.2 Research Question 2
- 1.3 Related Works 3
- 1.4 Motivation 3
- 1.5 Report Structure 4

2 Theory 5
- 2.1 Random Variables 5
- 2.2 Markov Property 6
- 2.3 Entropy 6
- 2.4 Information Sources 8
- 2.5 One Way Functions 8

3 Randomness 9
- 3.1 Random Generators 9
  - 3.1.1 Linux Kernel Random Generator 10
  - 3.1.2 Yarrow 12
  - 3.1.3 Fortuna 12
  - 3.1.4 `PRNGD` 13
  - 3.1.5 HAVEGED 14
- 3.2 Entropy Estimation 14
  - 3.2.1 Plug In Estimator 15
  - 3.2.2 Lempel–Ziv based estimator 15
  - 3.2.3 Context-Tree Weighting (CTW) 15
  - 3.2.4 Predictors 16
  - 3.2.5 Linux Kernel Estimator 17

4 Data Collection 19
- 4.1 Reference Generators 19
- 4.2 Linux Random Number Generator 21
- 4.3 FreeBSD Fortuna Generator 21
- 4.4 `HAVEGED` 22
- 4.5 Pseudo Random Number Generator Daemon (`prngd`) 22

5 Predictor Modifications **23**
5.1 Entropy Estimate . . . 23
5.2 Predictor Collections . . . 24
5.3 Transforming Input . . . 25

6 Implementation **27**
6.1 Data Format . . . 27
6.2 Estimations . . . 27
6.3 Markov Chains . . . 28
6.4 Collections . . . 28
6.5 Delta Transformation . . . 29
6.6 Other Implementations . . . 29
6.6.1 Lag Transformation . . . 29
6.6.2 Moving Window Transformation . . . 30
6.6.3 Function Transformation . . . 30
6.6.4 Choice Transformation . . . 30

7 Results **31**
7.1 Reference Generators . . . 31
7.2 Linux Entropy Estimation . . . 36
7.2.1 Timer Randomness . . . 37
7.2.2 Interrupt Randomness . . . 37
7.3 FreeBSD Entropy Estimation . . . 39
7.4 PRNGD Analysis . . . 41
7.5 HAVEGED Analysis . . . 42
7.5.1 Algorithm Analysis . . . 42
7.5.2 HAVEGED Entropy . . . 43

8 Discussion and Conclusions **44**
8.1 Reference Generators . . . 44
8.2 Linux Estimator . . . 45
8.2.1 Timings . . . 45
8.2.2 Interrupts . . . 46
8.2.3 Impact . . . 46
8.3 FreeBSD Generator . . . 47
8.4 PRNGD . . . 48
8.5 HAVEGED . . . 49
8.5.1 Knowledge of Timings . . . 50
8.5.2 Knowledge of State . . . 51
8.5.3 No Previous Knowledge . . . 52
8.5.4 Impact . . . 52
8.6 Predictor Modifications . . . 53
8.7 Final Conclusions . . . 53
8.8 Future Work . . . 54

Bibliography **56**

A ChaCha20 Generator in Linux **59**

B HAVEGED code 61

Glossary

CSRNG Cryptographically Secure Random Number Generator.

IID Independent and Identically Distributed.

MCW Most Common in Window.

PRNG Pseudo Random Number Generator
Random number generator creating data which appears random without need for as much actual random data. Usually works by expanding an unpredictable seed into more data..

RdRand Instruction available on certain Intel CPUs which return random numbers.

RNG Random Number Generator.

SHA Secure Hashing Algorithms,
a family of cryptographic hash algorithms.

TRNG True Random Number Generator
Random number generator which gathers true randomness and transforms to uniformly distributed random numbers. Transformations on numbers are used to give them better distribution but will not output more data than it receives..

xor Exclusive or.

Chapter 1

Introduction

This chapter introduce the background to the thesis and define the problem that is analysed during the rest of this report.

1.1 Background

Random numbers are necessary in several contexts such as simulations, games and cryptographic protocols. In most cases it is sufficient for the random numbers to pass statistical tests attempting to distinguish them from uniformly distributed random numbers. In cryptography however, passing statistical tests is not enough. Random numbers are necessary to secure several cryptographic protocols and if they are predictable, the security of the protocols would be threatened. For example, keys used to encrypt messages are required to be hard for an attacker to guess. Predictable keys means that an attacker can guess each bit of the key with better success probability than 50% which weakens the security of the key. Because of this, random data which is used in cryptographic applications needs to be hard to predict by an attacker, even if the attacker knows how it was generated. There are deterministic cryptographically secure random number generators which, when given a seed as input, gives an output which is provably hard [1] to distinguish from a real random sequence if the seed is unknown. Therefore, it is required that the seed used to generate the random data is in fact random to an attacker, as the attacker otherwise can guess the seed to break the number generator. In order to get an unpredictable seed it is necessary to have some non–deterministic process which can produce unpredictability. This unpredictability can then be conditioned into something which is uniformly distributed and then be used as a seed for a deterministic, cryptographically secure random number generator. Some simple examples giving unpredictable seeds are to simply flip a coin or roll a die enough times to produce a long seed. For generators on home computers which only need to be seeded once, this approach is feasible. Several small devices that all need seeds or multiple virtual machines running on the same computer are however environments where it quickly becomes infeasible to manually generate all seeds.

To determine the amount of data needed for sufficient security, a measure of the unpredictability a process produces is useful. This is available as the entropy [2] of the data, which directly corresponds to the amount of unpredictability. Entropy of data is easily computed if the distribution of the random data is known. In most use cases, the distribution is however unknown and thus another way of estimating the entropy is needed. This is in practice done by estimators which estimate the entropy of sampled data in various ways. The perhaps easiest estimator of entropy in data is the one which simply estimates the probabilities based on the relative frequencies in the observed data. This estimator is for example used in the *ent* test suite [3]. With enough analysed data and with the samples behaving as if Independent and Identically Distributed (IID), this estimation could be considered good. If the estimator is supposed to give better estimates with less available data or give estimates in real-time, meaning estimating the entropy for each sample when it receives it, then more advanced techniques are necessary [4]. Furthermore, some operating systems, such as Linux, also need an entropy estimator to estimate when random numbers can safely be generated unpredictably. In this application it is also necessary for the estimator to be very efficient, as it is executed often and should not impact the performance of the rest of the operating system [5, 6]. Another important property for estimators used in practice is that they do not overestimate the input entropy, as overestimates could lead to security vulnerabilities.

Even more difficulty arises when the samples cannot be assumed to be IID. To correctly estimate the entropy for general distributions is computationally infeasible but some estimates can still be achieved. The NIST Special Publication 800-90B [7] have several estimators which are used to estimate the entropy of non-IID data. These contain estimates based on common values, collisions, Markov chains and more. They also include a different type of estimators which instead of trying to estimate the entropy tries to predict the samples. This approach gives overestimates of the entropy as it is based upon actual predictions and thus never underestimates predictability [8]. This is the opposite of the requirement for estimators in entropy sources as underestimates are desired in order to not risk weakening the security. However, it is in practice impossible to guarantee that an estimator always provides a non–zero underestimates of entropy for arbitrary entropy sources while overestimates are possible. This makes overestimates useful as they bound the entropy rate of sources while alleged underestimates are unable to say anything with certainty.

There is also several test suites such as *dieharder* [9] and *TestU01* [10] which distinguish between uniform random distributions and other types of data. These are however unable to say to what extent the data is usable as a source of unpredictability if it is not uniformly distributed. As such these tests will only allow discarding of data which is not uniformly random, although this data could still potentially contain unpredictability.

1.2 Research Question

The research question answered in this is to investigate entropy usage in random number generators as well as how this can be analysed.

This question was investigated through two separate but connected goals. The first goal was to develop methods that estimate entropy for different sources of data. These methods were meant to provide overestimates of Shannon-entropy in expectation for non IID data, allowing them to be used on all kinds of data. This was done on both real world data and simulated data. The simulated data have known probability distributions and thus a known entropy which the results of the estimators can be compared to. This gave a notion of how well the estimators functioned on different types of distributions.

The second goal relates to the real world generators analysed with the developed method. Results from entropy estimation of the generators allows an analysis of their functionality in regard to entropy input. Together with an analysis of the source code of the generators this gives a way to potentially detect weaknesses in how they deal with entropy. Therefore, the second goal more directly works towards the question of investigating entropy usage in random number generators.

1.3 Related Works

Research investigating several ways of estimating entropy has been done. These include approaches based on methods similar to well–used compression algorithms [11], methods which estimate min-entropy [7] as well as methods based on predicting the data [8]. There is also other estimations which are used in actual entropy sources in order to estimate the entropy rate of the source. An example of such a source is the Linux kernel random number generator [12]. This generator and its entropy estimation have been analysed multiple times [13, 5, 14, 15]. The entropy estimator in the kernel is required to be both space and time efficient and also give underestimates of the entropy. Because of this the estimator have to be relatively simple and make conservative estimates of the amount of entropy in samples. The construction of the estimator is somewhat arbitrary but Lacharme et al. [5] shows that it seems to behave as expected. Some research has been done on the random number generator available in the Windows operating system [15, 16] but other implementations of generators with entropy input have been analysed less.

1.4 Motivation

There is no previous in-depth comparisons of the Linux entropy estimates and other estimates for the same data. This analysis could be of interest as the Linux estimate is an approximation, meant to hopefully underestimate the entropy per sample, while outside estimates can provide guaranteed entropy overestimates. If an outside estimator manage to provide an entropy overestimate that is lower than the alleged underestimate of the generator, a case where the generator overestimates entropy is found. That the generator overestimates entropy could potentially lead to security vulnerabilities. Looking for such overestimates may thus potentially discover vulnerabilities in the generator which previously were unknown.

Entropy estimators which were previously available are however not optimal for such an analysis. Most available methods only give an average entropy per sample for the whole source. Such an estimate will probably not be low enough to detect problems where the generator estimate occasionally is higher than the actual entropy. Other methods which estimate min-entropy are able to detect local behaviour where the source produces less entropy. This is however not that interesting without comparing it to the estimate made by the kernel, where periods of relative predictability are expected and accounted for. An estimator which provides overestimates of Shannon-entropy rate for arbitrary sources would however be useful for such an analysis, but no good alternative exist.

With a newly developed method available for estimating entropy, it is also easy to analyse entropy used by other generators. As analysis of entropy usage in generators have been very limited, this can potentially lead to new vulnerabilities being discovered in these generators.

1.5 Report Structure

This thesis begins with background theory in Chapter 2. Following is background of random generators with entropy input used in practice, combined with information about how generators deal with entropy estimation and some examples of entropy estimators in Chapter 3. Afterwards follows descriptions of how entropy was collected from different generators in Chapter 4. This is followed by a description of theory and implementation of the constructed method to estimate entropy in Chapters 5 and 6. The results of using the new estimator and results related to the entropy generators are then presented in Chapter 7. Finally, a discussion about the results and impact for the different generators is presented in Chapter 8.

Chapter 2

Theory

This chapter will introduce some basic theory which may be useful in the rest of the thesis.

2.1 Random Variables

A discrete random variable $\boldsymbol{X}$ is related to a set of possible outcomes A for the random variable. For each possible outcome $x_i \in A$, the probability that the random variable takes that outcome is denoted $P(\boldsymbol{X} = x_i) = p(x_i)$. The expected value of the random variable is defined in Formula 2.1 and intuitively corresponds to the average value of observations will converge to when the number of samples increases [17].

$$E(X) = \sum_{x_i \in A} p(x_i) x_i \tag{2.1}$$

Given two random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$, with outcome sets A and B respectively, the probability of $\boldsymbol{X}$ to be $x_i \in A$ and $\boldsymbol{Y}$ to be $y_j \in B$ is denoted $P(\boldsymbol{X} = x_i, \boldsymbol{Y} = y_j)$. The variables are independent iff $P(\boldsymbol{X} = x_i, \boldsymbol{Y} = y_j) = P(\boldsymbol{X} = x_i)P(\boldsymbol{Y} = y_j)$. For random variables which are not independent it can be interesting to know the distribution of one of the variables when the outcome of the other is known. This conditional probability for outcome $\boldsymbol{X} = x_i$ when the outcome $y_j = \boldsymbol{Y}$ is known, is denoted $P(\boldsymbol{X} = x_i \mid \boldsymbol{Y} = y_j)$ and can be calculated with Formula 2.2. For independent variables, the conditional probability is the probability of the variable itself [18] $P(\boldsymbol{X} = x_i \mid \boldsymbol{Y} = v_j) = P(\boldsymbol{X} = x_i)$.

$$P(\boldsymbol{X} = x_i \mid \boldsymbol{Y} = y_i) = \frac{P(\boldsymbol{X} = x_i, \boldsymbol{Y} = y_j)}{P(\boldsymbol{Y} = y_j)} \tag{2.2}$$

A set of random variables $\{\boldsymbol{X}_i\}$ is said to be IID if they are independent and it holds that $P(\boldsymbol{X}_i = x_j) = P(\boldsymbol{X}_k = x_l)$ for all i, j, k, l. A sequence $s_0, s_1, s_2, \ldots, s_N$ of outcomes of the IID random variables is such that the outcome of $\boldsymbol{X}_i$ is $s_i = x_j$ for some $x_j \in A$ for all i. Given such a sequence of the IID random variables the mean μ^* of the outcomes may be calculated. This is the outcome of another random variable $\boldsymbol{M}$ defined in Equation 2.3.

$$M = \sum_{i=0}^{N} \frac{\boldsymbol{X}_i}{N} \tag{2.3}$$

It is then possible to show that $E(\boldsymbol{M}) = E(\boldsymbol{X})$ if $\boldsymbol{X}$ is another IID copy of the random variables. This means that the average of multiple observations of the same variable has the same expected value as the random variable.

To determine how quickly the average of the observations converge to the expected value the variance of the random variable is useful. This is defined through the formula $V(\boldsymbol{X}) = E[(\boldsymbol{X} - E(\boldsymbol{X}))^2]$ which can be shown to equal $E(\boldsymbol{X}^2) - E(\boldsymbol{X})^2$. This measure corresponds to how much the observations are expected to differ from the mean of the observations. It can then be seen that for N IID random variables it is possible to show that $V(\boldsymbol{M}) = V(\boldsymbol{X})/N$. This shows that the variance of the mean decrease when more observation are used and gives a way to determine how quickly the mean converges to the expected value of the random variables [19].

2.2 Markov Property

A sequence of random variables is Markovian if the probability distribution of the upcoming variable only depends on the current state and not how this state was reached. The simplest way this can be realized is with the state directly corresponding to the outcome of the previous random variable. For random variables $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_N$ this can be expressed as in Equation 2.4 and thus the outcome of $\boldsymbol{X}_i$ is independent of the results of $\boldsymbol{X}_j$ for $j < i - 1$ if the outcome of $\boldsymbol{X}_{i-1}$ is known.

$$P(\boldsymbol{X}_i \mid \forall j < i : \boldsymbol{X}_j = s_j) = P(\boldsymbol{X}_i \mid \boldsymbol{X}_{i-1} = s_{i-1}) \tag{2.4}$$

However, the variables $\boldsymbol{X}_i$ and $\boldsymbol{X}_j$ are not necessarily independent if $j < i - 1$ as the probability $P(\boldsymbol{X}_i \mid \boldsymbol{X}_j)$ can be expressed as in Equation 2.5 which in general is not equal to $P(\boldsymbol{X}_i)$.

$$P(\boldsymbol{X}_i \mid \boldsymbol{X}_j) = \prod_{k=j+1}^{i} P(\boldsymbol{X}_k \mid \boldsymbol{X}_j, \boldsymbol{X}_{j+1}, \ldots, \boldsymbol{X}_{k-1}) = \prod_{k=j+1}^{i} P(\boldsymbol{X}_k \mid \boldsymbol{X}_{k-1}) \tag{2.5}$$

A generalization of this is to instead allow the state of the system to be decided by N previous outcomes of random variables. This can be expressed as

$$P(\boldsymbol{X}_i \mid \forall\, j < i : \boldsymbol{X}_j = s_j) = P(\boldsymbol{X}_i \mid \boldsymbol{X}_{i-1}, \ldots, \boldsymbol{X}_{i-N}) \tag{2.6}$$

and can be shown identical to the first definition by defining $Y_i = (\boldsymbol{X}_i, \boldsymbol{X}_{i-1}, \ldots, \boldsymbol{X}_{i-N+1})$ which gives equation 2.7.

$$.P(Y_i \mid Y_{i-1}) = P(\boldsymbol{X}_i, \boldsymbol{X}_{i-1}, \ldots, \boldsymbol{X}_{i-N+1} \mid \boldsymbol{X}_{i-1}, \ldots, \boldsymbol{X}_{i-N}) = P(\boldsymbol{X}_i \mid \boldsymbol{X}_{i-1}, \ldots, \boldsymbol{X}_{i-N}) \tag{2.7}$$

and thus this dependence of length N can be expressed as an ordinary Markov chain depending only on the outcome of the previous random variable [20].

2.3 Entropy

In statistical physics entropy is a measure of order in systems. Information theory also deal with the amount of order or uncertainty in a system. An analogous notion of entropy for information theory was thus introduced by Shannon [2]. The entropy for a random variable $\boldsymbol{X}$ with outcomes $x_j \in A$, each with probability $P(\boldsymbol{X} = x_j) = p_j$ is defined as in Equation 2.8 which have afterwards been named Shannon entropy.

$$H(\boldsymbol{X}) = -\sum_j p_j \log p_j \tag{2.8}$$

The logarithm could be in any base, but is most commonly used with base 2 in which case the entropy is measured in bits. Systems with n bits of entropy can in theory be optimally compressed into approximately n bits of data describing the system. There are other definitions of entropy which give measures useful for other applications. For this thesis, one of these definitions which is of interest is the min-entropy H_∞, which is the negative logarithm of the p_j with maximal probability, as shown in Equation 2.9.

$$H_\infty = \min_{x_j \in A}(-\log P(\boldsymbol{X} = x_j)) = -\log(\max_{x_j \in A} P(\boldsymbol{X} = x_j)) \tag{2.9}$$

This is of interest in security applications where the worst case scenario is interesting, which in the case of random numbers correspond to the most probable value. For actual applications it is also a goal to not overestimate the entropy, as this could lead to weaknesses. Therefore, an estimate of the conservative min-entropy can be used. It is however computationally infeasible, if not even impossible, to always provide a non–zero underestimate of entropy for general sources.

When data consists of a sequence of several samples from their own distributions, the entropy per sample is a relevant measure. If the distributions are identical and independent then the entropy for each sample will be the same. As the entropy of two independent events following each other is additive [2] this gives that the entropy for all the data simply is the number of samples times the entropy per sample. More generally the entropy for two events $\boldsymbol{X}$ and $\boldsymbol{Y}$ can be expressed as $H(\boldsymbol{X}, \boldsymbol{Y})$ which is defined in Equation 2.10 where $p(i, j)$ is the probability of $\boldsymbol{X} = x_i$ and $\boldsymbol{Y} = y_j$. This is equal to $H(\boldsymbol{X}) + H(\boldsymbol{Y})$ if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent. Introducing the conditional entropy $H_{\boldsymbol{X}}(\boldsymbol{Y})$ defined in Equation 2.11 where $p_i(j) = p(j \mid i)$ is the conditional probability of $\boldsymbol{Y} = y_j$ given $\boldsymbol{X} = x_i$. This value measures the entropy of $\boldsymbol{Y}$ when the outcome of $\boldsymbol{X}$ is known. It is then easy to show that for any two events Equation 2.12 holds, even if the events are not independent [2].

$$H(\boldsymbol{X}, \boldsymbol{Y}) = \sum_{i,j} p(i, j) \log(p(i, j)) \tag{2.10}$$

$$H_{\boldsymbol{X}}(\boldsymbol{Y}) = \sum_{i,j} p(i, j) \log(p_i(j)) \tag{2.11}$$

$$H(\boldsymbol{X}, \boldsymbol{Y}) = H(\boldsymbol{X}) + H_{\boldsymbol{X}}(\boldsymbol{Y}) \tag{2.12}$$

Another measure related to entropy is the Kullback-Leibler divergence, also called relative entropy. Denoted by $D(p||q)$ for two probability mass functions p and q with possible outcomes in A it is calculated as in Equation 2.13. This measure is not an actual distance as it neither is symmetric nor fulfils the triangle inequality. It is however true that $D(p||q) \geq 0$ with equality if and only if $p = q$. The measure may however still be thought of as the "distance" between two distributions [21].

$$D(p||q) = \sum_{x \in A} p(x) \log \frac{p(x)}{q(x)} \tag{2.13}$$

2.4 Information Sources

When analysing entropy estimators it may be possible to give certain guarantees on the performance, depending on the type of source. Three properties of sources which may be interesting are whether they are memoryless, ergodic or stationary.

A memoryless information source is a source which fulfils the restrictive requirement that the source have no memory of previous events, and thus cannot be influenced by them. This is thus identical to the samples coming from independent and identically distributed random numbers [22]. A less restrictive requirement on the source is that it is ergodic. An ergodic process is such that a single sufficiently long sequence of samples is always enough to determine the statistical properties of the source [23]. An example of a non-ergodic source is the source which first flips a coin and have the first output 0 or 1 depending on the outcome, while all following outputs simply equals the first output. This is non-ergodic as all samples in a single sequence will all equal 0 or 1 while multiple tests of this source will give half 0 and half 1.

A stationary source is one which has a probability distribution which does not change over time. This includes all sources which generates independent samples. However, sources may also generate samples that depend on each other but where the distributions are stationary. As such, stationary source do not necessarily have to generate independent samples.

2.5 One Way Functions

One way functions are easy to compute functions that are hard to invert. Such functions are tightly coupled with pseudo random number generators and an actual PRNG is possible if and only if one way functions exist [1]. In practice there are several functions, which although not proven one–way, have no known way in which they can be inverted efficiently. Examples of such function include cryptographically secure hash–functions such as different versions of SHA. These hash–functions are also used in some applications to generate random looking data.

Chapter 3

Randomness

This chapter deals with randomness and entropy estimations which can be used to approximate the amount of uncertainty in random data.

3.1 Random Generators

In several contexts, such as simulations, games and cryptography it is necessary to have random numbers for various purposes. However, a computer is deterministic and is therefore bad at providing random data by itself. One solution to this is external devices that provide unpredictable data from non–deterministic processes. Another solution to this is to use a PRNG which deterministically can produce data which appears random. The fact that the data appears random means different things in different contexts. In simulations, it is important that the data is unbiased and passes statistical tests which try to distinguish the data from a uniform random distribution. For cryptography, it is also required that an attacker should be unable to predict the data, even if the type of generator is known. Non–deterministic random data is thus required to make the generator unpredictable to attackers. This initial randomness can be used as a seed to the generator. If the generator is cryptographically secure it is then possible to guarantee, under certain assumptions, that an attacker cannot distinguish the output from a uniform distribution in polynomial time in the size of the seed [1]. This allows a small amount of actual random data to be expanded into an arbitrary amount of cryptographically secure random data.

Furthermore, it is beneficial if a cryptographically secure random number generator is backward and forward secure. These properties are related to what security guarantees can be given when output r is generated from an RNG at time t. Forward security means that an attacker who compromises the state at time $t' > t$ should be unable to predict r meaning that numbers generated are safe against attacks forward in time. Backward security is similarly that an attacker who compromised the state at time $t' < t$ should be unable to predict r. Number generated thus remain safe against attacks backward in time. To provide forward security the generator can use one–way functions which are believed to be hard to invert. By altering the state periodically with such a function, a compromised state will not leak information about previous states unless the one–way function is inverted. This provides forward security without requiring more external randomness. Backward security do however require that more external randomness is introduced to the generator. This is necessary as the generator otherwise is completely deterministic and all future states are then easily predicted after an initial state becomes known [24].

A constant stream of entropy is thus required in order to guarantee backward security. This entropy can come from hardware specific for the purpose [25], software which gather uncertainty from timings or states in the computer [26, 27] or even websites which provide random numbers based on atmospheric noise [28].

The amount of entropy added to the generator is also required to be large enough when used to generate new random data. An attacker who at one point have compromised the generator can otherwise potentially guess future states of the generator. When the added entropy is low, the possible states the generator can be in will be low. With knowledge of some actual outputs x_t of the generator the attacker may be able to identify which of his guesses give the same output. In fact, it may even be enough for the attacker to be able to monitor output of some function $f(x_t)$ of the data. An attacker may thus potentially recover the state when only one of the guessed states is able to produce the given output. This allows the attacker to perform an iterative guessing attack where knowledge of the generator state is kept even when entropy is added after it was compromised [29].

Examples of how actual generators are constructed and how they deal with these problems are presented next.

3.1.1 Linux Kernel Random Generator

The Linux kernel contains a random number generator with entropy input which have interfaces to both a PRNG (/dev/urandom) and a TRNG (/dev/random). The TRNG attempts to ensure that it gathers at least as much entropy as it outputs random data. This is done by blocking until it estimates that enough entropy has been gathered. The PRNG can always generate random numbers without blocking, using some gathered entropy but without requirements on amount of estimated entropy.

The TRNG is based on entropy pools, fixed size memory areas where unpredictable data is stored. Each entropy pool also have a counter which keeps track of estimated contained entropy. Outside entropy is added to one of the pools, the input pool, while the other pool, the output pool, is used to generate data. Added entropy is mixed with the input pool using a mixing function and the entropy counter is incremented by a conservative estimate of the added entropy. Upon output requests, data is attempted to be generated from the contents of the output pool directly. However, the output pool entropy counter is often close to 0 meaning that entropy must be requested from the input pool. If the input pool has enough contained entropy, an output from this pool, approximately equal in size to the requested amount of entropy, is generated. This data is mixed into the output pool and the entropy pool counters are changed by the amount of data transferred. After the transfer, the output pool have enough contained entropy to provide the output to the user. After output is generated, the output pool entropy counter is decreased by the amount of data generated. To generate output the SHA–1 hash of the whole pool is calculated, with output being this hash folded in half. This means that the output is the exclusive or of the first and second halves of the 160 bit hash. The hash is also mixed back into the pool. If both pools lack entropy, the TRNG will not provide output until enough entropy is gathered.

The non–blocking generator will generate random numbers even if the estimated en-

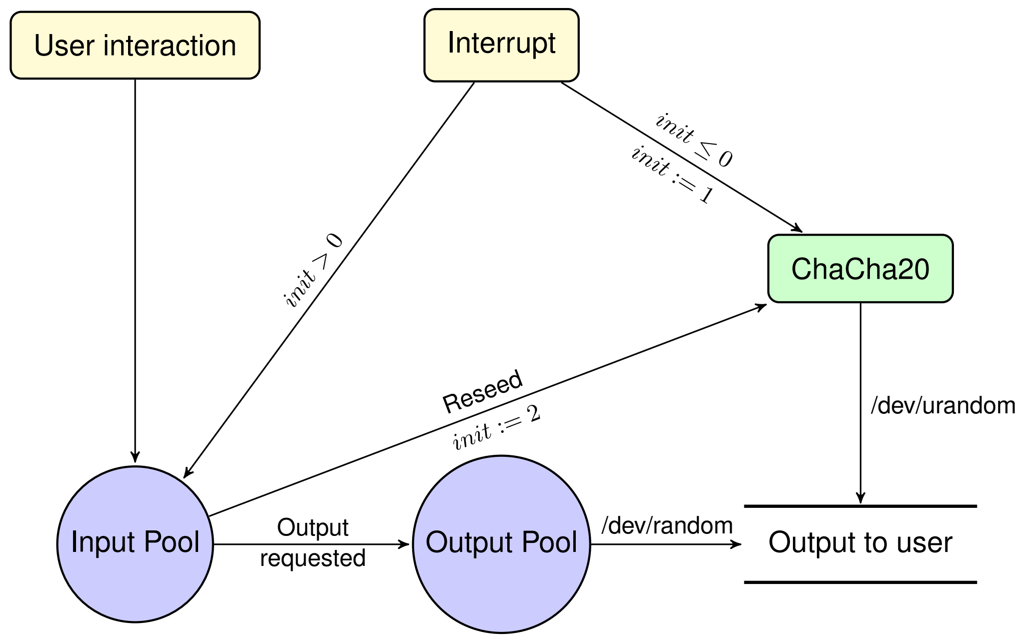

Figure 3.1: Diagram over how entropy is transferred in the Linux kernel generator

tropy is low. Previously this generator worked on a similar pool concept as the blocking generator. However, this changed in June 2016 [30]. It now works more like a traditional PRNG and uses the ChaCha20 algorithm to generate random data. This generator is reseeded approximately every 5 minutes and after first initialized, will no longer care about entropy estimates. The first seeding is however impacted by the entropy estimation and will not occur until it is estimated at least 128 bits of entropy have been gathered. Furthermore, a "fast load" of the generator may be performed. If hardware random number generators are added, they are used for this. Otherwise, when the estimated amount of entropy gathered from interrupts is at least 64 bits, the generator is initialized with this entropy. It is however still not considered fully seeded and will automatically reseed when the input pool contains at least 128 bits of entropy. Figure 3.1 shows how this works with an $init$ variable corresponding to level of initialization of the ChaCha20 generator. Reseeding of the ChaCha20 algorithm is thus performed when $init < 2$ and the input pool entropy counter is greater than 128 bits. This is also done approximately 5 minutes after the last reseed.

To provide backtrack protection the generator also mutates the key with any remaining output from ChaCha20 which has not been given to the user [31, 5]. Furthermore, as the generator is reseeded every 5 minutes, forward security over intervals longer than 5 minutes is provided. For more details about the ChaCha20 algorithm in the Linux kernel, see Appendix A.

Entropy gathered by the Linux generator comes from several sources. Any available hardware random number generators will be used for entropy. Other sources are always available such as disk–timings, user interactions and interrupt timings which, to varying extent, are unpredictable. Entropy contained in these events is unknown but is estimated in different ways, described in Section 3.2.5. Events are used in two different ways, with user-interactions and disk–events treated in one way and interrupts in another.

In the case of disk–events and user–interaction, the generator receives the time of the event as well as a code related to type of event. The time of an event is measured in *jiffies*, a coarse time unit used in the kernel corresponding to number of interrupts that have occurred since start–up. A third value is also included which is received through a call to the function `random_get_entropy`. Standard behaviour of this function is to get a value corresponding to the processor cycle count. This is however not possible for all architectures and for some architectures this function behave differently, with some configurations resulting in it always returning 0. The resulting sample containing these three elements is mixed into the input pool, trying to add some unpredictability.

For interrupts the process is a bit different. The input is similar, with the interrupt request that occurred and flags for the interrupt instead of an event code. Current time in jiffies as well as the data returned from the function `random_get_entropy` is also used. The data does not come from all interrupts in the kernel, with predictable interrupts not being used. The data is not directly added to the input pool and is instead kept temporarily in a "fast pool". This pool is 128 bits large and stores the entropy until it is estimated to be 64 bits of entropy or it has been approximately 1 second since the last time it was emptied. When emptied, this "fast pool" is mixed into the input pool.

3.1.2 Yarrow

Another algorithm for a random number generator with entropy input is available with the Yarrow generator [32]. This generator has two entropy pools and one PRNG that is regularly reseeded from the pools. This is done in order to ensure that its internal state is kept unpredictable, even if at one point compromised. The amount of entropy being added to the generator is estimated by taking the minimum of three different estimates. These estimates are an estimate provided by the source, a statistical estimate of entropy and the size of the collected data times a constant factor (less than 1). The entropy pools consist of a running hash of added entropy and a counter for how much entropy is estimated to be contained in the pool. Which pool to add input entropy to is alternated between the two and the difference between them is only how often they are used for reseeds. The so called fast pool is used more often than the slow pool which is meant to make more conservative reseeds for the PRNG to ensure that iterative guessing attacks become impossible.

In practice the fast pool is used for reseeds as fast as the estimated entropy contained is greater than 100 bits at which point a new seed is constructed from the pool contents and the old seed. After reseeding, the contained entropy in the pool is set to 0. The slow pool is instead used when at least $k < n$ of the n entropy sources have contributed at least 160 bits of entropy to the pool. When reseeding from the slow pool, the new seed is constructed from the contents of both the slow and the fast pool as well as the previous seed. After reseeding with the slow pool, the entropy counters of both pools is set to 0.

3.1.3 Fortuna

The Fortuna algorithm was developed in order to circumvent the problem of entropy estimation by completely bypassing it. The generator works with a block cipher to generate random data from a seed and a counter. Which block cipher to use is up to actual implementation but the designer recommends AES, Serpent and Twofish. The counter is

128 bits long and will be incremented each time random data is generated or the generator is reseeded. The key to the block cipher is 256 bits long and is constantly reseeded.

Unpredictability in the seed can be guaranteed to eventually return after a state compromise. This is guaranteed by keeping several entropy pools where some are used for frequent reseeds, while others are used less often. By alternating between the pools when adding entropy the entropy will increase in all the pools approximately evenly. This allows the slower pools to eventually gather enough entropy to become unpredictable to attackers. This means that reseeds with at least n bits of entropy will happen before a constant factor times n total bits of entropy has been gathered. Therefore, the problem of entropy estimation in the generator is removed. This construction can without entropy estimation guarantee that there eventually is a new seed which will be hard to guess for an attacker performing an iterative guessing attack.

In practice Fortuna has 32 entropy pools where pool i is only used for reseeds which divide 2^i. As such, pool 0 is used in every reseed while pool 31 is only used once every 2^{31} reseeds. By limiting the number of reseeds to at most once every 0.1 seconds the potential pool 32 would only be used approximately once every 13 years, and thus it is considered sufficient to have only 32 pools [33].

3.1.4 `PRNGD`

The `PRNGD` generator is a RNG meant to function as the UNIX non–blocking random number generator `/dev/urandom` on systems without this interface. It gathers entropy by calling system programs whose output to varying degrees is unpredictable. There is also another program for a similar purpose, namely `EGD` which also works by calling system programs for entropy. The `EGD` generator is however meant to be a replacement for the UNIX blocking RNG `/dev/random`, instead of `/dev/urandom`, meaning that they function differently when entropy estimates are low.

The programs the generator uses to gather entropy from is configurable in a configuration file. For each source in the file a number related to how much entropy the output the program is supposed to be worth is also included. This is expressed in how many bits of entropy is contained in each byte of output from the program. This fraction has to be manually configured for each program which requires knowledge of expected outputs and their unpredictability. The program keeps track of the contained entropy with a counter while the SHA-1 hash of the added data is added to a pool of size 4100 bytes. To add the data the Exclusive or (xor) of the hash and a part of the pool replaces the original data at that part of the pool and the entropy counter is incremented. A variable also keeps track of whether the generator can be considered seeded yet. Before a certain amount of entropy has been gathered, the pool is considered unseeded while after that, the pool will be seeded. Afterwards, pool will still be considered seeded no matter how much the entropy counter decreases.

When generating random numbers, the generator first checks that the pool has been seeded. If it has not yet been seeded it will not generate any output data. A seeded generator will generate random data by first mixing the entropy pool. Mixing is done by computing the SHA-1 hash of the current block of data in the pool and mix it into the next block of data in the pool with xor. This is done for all the blocks in the pool while wrapping around to the start of the pool when reaching the end position. Output random data is then computed as the SHA-1 hash of the current position of the pool. The position is then moved to the next block and the output hash value is mixed back into the pool. This is repeated until the requested amount of random data is retrieved. Afterwards, the pool is once again mixed as was done before the random data was generated. After the data is generated, the entropy counter is also decreased by the amount of generated data.

The main use of the entropy counter in the generator is thus to monitor when the generator can be considered seeded. The entropy counter is also used by the generator to determine how often to call the external programs. When the counter is decreased below a certain threshold, the generator will continuously call the gathering programs until the entropy counter is over the threshold. When the entropy counter is above the threshold the gathering programs will still be called, although this is done less frequently [27].

3.1.5 HAVEGED

The `HAVEGED` generator collects entropy from running time variations in a PRNG [34]. This generator is constructed in such a way that the execution time will vary based on the processor state. The number of cache hits or misses as well as the branch prediction of the processor causes run time variations. Furthermore, interrupts during the code execution will also add unpredictability to the execution time.

The PRNG expands these gathered execution times into a sequence of uniformly distributed random numbers. This is done with a large internal state, with a size that varies depending on cache size. Most of the state is a large array of values which should be approximately uniformly distributed. Two random walks through this array are performed in parallel with the walks decided by the array contents. During the walks, memory close in the array related to the current positions is updated with the current time, causing running time of the walks to impact the state. As the size of the array is based on the cache size, a certain percentage of cache–misses is expected when accessing the array. The generator also contains a variable based on high order bits of the position in one of the random walks. As such this variable should be uniformly distributed if the position in the array is. This variable is then tested in multiple statements exercising the computers branch prediction. As the value should be random, it should fail approximately half the time. Output from the generator is contents of the array close to the current positions in the random walks. In total the generator reads the time twice for every 16 output data samples [34, 26, 35].

3.2 Entropy Estimation

The formulas for Shannon– and min–entropy both require known probability distributions. In practice, the distribution for an entropy source may however be unknown. As such, entropy has to somehow be estimated. This can be done in several ways, for example by producing an estimated probability distribution or by calculating the entropy based on its connection with compression of data. Demands on estimators also varies depending on context. Estimators in generators must be efficient and work in real–time, other estimators can be less efficient and work with all the data to analyse the properties of a source.

3.2.1 Plug In Estimator

A simple entropy estimate for IID samples is to estimate the probabilities of samples with their relative frequencies. These estimates $\hat{p}(x_j) = n_j/n$, given a total of n samples with n_j of them being equal to x_j, can then be used in the formula for entropy. This gives the plug in estimate of entropy in Equation 3.1.

$$\hat{H}(\boldsymbol{X}) = -\sum_{x_j \in A} \hat{p}(x_j)\log(\hat{p}(x_j)) \tag{3.1}$$

The values of $\hat{p}(x_j)$ are the maximum likelihood estimates of the probabilities $P(\boldsymbol{X} = x_j)$ for memoryless sources and gives a universal and optimal way of estimating the entropy for identical independent distributions, as well as for sources with finite alphabets and finite memory [36].

3.2.2 Lempel–Ziv based estimator

The Lempel–Ziv based estimators work on the same idea as the compression algorithm invented by Lempel and Ziv. This compression algorithm keeps a moving window of observed samples in memory. When a sample is read, the longest sequence in memory which is identical to the most recently read samples is found. Data is then encoded by distance to the match as well as match length and the final sample in the match [37].

The same idea can give formulas that tend to the entropy of stationary ergodic information sources as different parameters tend to infinity. For example for an ergodic information source $\{\boldsymbol{X}_k\}_{k=-\infty}^{\infty}$ for some outcomes $\forall k\ \boldsymbol{X}_k = s_k$ the parameter $\tilde{N}_l$ is defined as the smallest $N > 0$ satisfying Equation 3.2. This parameter then gives the Formula 3.3 which is true in probability [11].

$$(s_0, s_1, \ldots, s_{l-1}) = (s_{-N}, s_{-N+1}, \ldots, s_{-N+l-1}) \tag{3.2}$$

$$\lim_{l\to\infty} \frac{\tilde{N}_l}{l} = H(\boldsymbol{X}_k) \tag{3.3}$$

Using this and similar limits it is possible to provide entropy estimates which are guaranteed to converge to actual entropy as parameters increase towards infinity.

3.2.3 Context-Tree Weighting (CTW)

The context tree weighting method combines several Markov models of different orders to estimate entropy. To do this the model uses multiple different D–bounded models. These models are sets of sequences with length no longer than D together with a probability distribution for elements which could follow every member of the set. The set S is also required to be a complete and proper suffix set, meaning that all strings x have a unique suffix s in S. A suffix s with length l of a string x is here such that the l last characters of x are the characters of s.

The probability of a sequence is computed as a weighted sum of probabilities of the sequence from all possible D-bounded models. With the set of all D-bounded models called M_D the probability $\hat{P}_{\text{CTW}}(x_1^T)$ of a sequence x_1^T of length T is calculated as.

$$\hat{P}_{\text{CTW}}(x_1^T) = \sum_{M \in M_D} w(M) \cdot \hat{P}_M(x_1^T) \tag{3.4}$$

$$\hat{P}_M(x_1^T) = \prod_{i=1}^{T} \hat{P}_M(x_i \mid \text{suffix}_M(x_{i-D}^{i-1})) \tag{3.5}$$

Here $w(M)$ is the weight of the model M and $\text{suffix}_M(x_{i-D}^{i-1})$ is the unique suffix in M to the sequence x_{i-D}^{i-1}. The weights $w(M)$ and the probabilities $\hat{P}_M(x \mid q)$ are decided cleverly to ensure that the sum over all possible models can be computed efficiently [38]. Details of how this is done are not further explained here.

The methodology with context-tree weighting was originally intended for data compression. However, it can easily be adopted to entropy estimation via Equation 3.6 which can be shown to provide overestimations of the entropy rate of the source. The method's performance as an entropy estimator has also been tested and compared to LZ-based estimators with the CTW estimator performing best [39].

$$\hat{H} = -\frac{1}{n} \log(\hat{P}_{CTW}(x_1^n)) \tag{3.6}$$

3.2.4 Predictors

Another approach to entropy estimation is to try to predict the data. If N samples are read and n are successfully predicted, then the success rate $p = n/N$ can be used to estimate the min-entropy. This estimate is the negative logarithm of this p [8].

Several predictors can be constructed, with predictors that are good matches for the data correctly predicting more samples. This means that better matches cause lower entropy estimates, while still being overestimates. This allows the approach of using multiple predictors on the same data and then use the lowest of the estimates they provide. With this approach it suffices that any one predictor match the source in order for its lower estimate to be used [8].

Multiple different predictors have been constructed with different approaches. Some of these are

Most Common in Window (MCW)
: Constructed with a parameter w and keeps a moving window of fixed size of w samples with previously read data. Uses the most common value in the window as prediction.

Lag predictor
: Constructed with parameter N and keeps the N last seen values in memory. Predicts the sample that appeared N samples back.

Markov Model with Counting (MMC)
: Constructed with parameter N and remembers all N-sample strings seen, keeping a count for each value that followed. Predicts the one which is most commonly seen after the currently read N-sample sequence.

LZ78Y
: Inspired by the idea behind the Lempel-Ziv compression. Keeps track of all observed strings of up to length 32 until its dictionary reaches maximum size. For each such string it keeps track of the number of every sample which followed the string. The prediction then depends on the previously read samples and the longest string in the dictionary which it matches. The actual prediction is then the most common value which has previously followed this string.

First order difference
: Only works with numerical data, guessing that difference between events is constant. More specifically, if the previous two numbers are x and y then it predicts that the next element should be $z = x + (x - y)$. Furthermore, it keeps track of previously observed numbers and rounds the value of z to the closest one of these previously seen numbers.

Furthermore, predictors may be combined into ensemble predictors. These consist of multiple predictors with the actual prediction depending on which predictor has the highest success rate. The ensemble predictors produced contain multiple of the same type of predictors, with different parameters. This is done for the MCW, Lag and MMC predictors where the parameters take on all possible values in a set. More specifically the MCW predictors take the parameters $w \in \{63, 255, 1023, 4095\}$ while the Lag and MMC models take on all values for N less than some chosen limit [8].

Another property the predictors look at is the local predictability of the source. As the min-entropy is estimated by the predictors, a high predictability of a local sequence should lower the entropy estimate. In order to determine local predictability the predictors analyse the probability of recurrent events. With r being one greater than the longest run of successful predictions this leads to Formula 3.7 for an upper bound on the success probability p. Here n is the number of predictions $q = 1 - p$ and x is the real positive root to the equation $q - x + qp^r x^{r+1} = 0$. Furthermore, α is the probability that there is no run of length r. The calculated p is thus the probability for which there is no success runs of length r or longer with probability α. As such, p is the highest probability such that the probability of there being no runs with length r is lower than α [8].

$$\alpha = \frac{1 - px}{(r + 1 - rx)q} \cdot \frac{1}{x^{n+1}} \tag{3.7}$$

3.2.5 Linux Kernel Estimator

To estimate unpredictability in input events the Linux kernel contains an entropy estimator. This estimator is required to be efficient and work in real–time, estimating entropy of samples as they become available. This is necessary as it is called every time new data is added to the entropy pool, which is often. Underestimates of actual entropy are also desirable, with too high estimates potentially weakening security of the system.

Entropy credited to the input pool from disk–timings and user–interaction is calculated only from the timing of the event, and with resolution in jiffies. The rest of the data, described in Section 3.1.1, is not used for entropy estimations but still provide unpredictability to the entropy pool. Estimation is done with a data structure for each type of event, storing some previous timings. The stored data is the time t of the last occurring event of this type, as well as δ, the difference in time between the previous event and the one before it. Furthermore, it stores δ_2, the difference between the previous δ and the one before that. As a new event at time t' occurs, the kernel calculates δ', δ_2' and δ_3' as below.

$$\delta' = t' - t \tag{3.8}$$
$$\delta_2' = \delta' - \delta \tag{3.9}$$
$$\delta_3' = \delta_2' - \delta_2 \tag{3.10}$$

The values of t', δ' and δ_2' are then used to update the stored values of t, δ and δ_2 respectively. The entropy estimate is based on d, the minimum of $|\delta'|$, $|\delta_2'|$ and $|\delta_3'|$. If $d = 0$ the estimate is zero while otherwise calculated as

$$\min(\lfloor \log_2 (d) \rfloor, 11) \tag{3.11}$$

The estimated entropy is thus limited to a maximum of 11 bits of entropy in order to ensure underestimates [5].

Input from interrupts has a simpler process, as less work can be done during interrupts. The entropy estimation is that all interrupt requests add 1 bit of entropy. In case the computer have an "architectural seed generator", it is also called each time interrupt randomness is added. This allows extra data to be added which the kernel estimates contains one extra bit of entropy.

Estimated entropy is not added to the counter keeping track of stored entropy directly as some previously stored entropy contents might be overwritten. It is claimed that new contributions to an entropy pool will approach the full value asymptotically. This is modelled through Formula 3.12 which estimates contained entropy in a pool with maximum size s that previously had p bits of entropy when a bits of entropy are added.

$$p + (s - p) \cdot (1 - e^{-a/s}) \tag{3.12}$$

Calculating this exponential is however considered too inefficient and it is instead approximated with the help of Equation 3.13 when $a \leq s/2$.

$$1 - e^{-a/s} \geq \frac{3a}{4s} \tag{3.13}$$

$$(s - p)\,\frac{3a}{4s} = \frac{3a}{4}\left(1 - \frac{p}{s}\right) \tag{3.14}$$

The pool counter is then updated with at most $s/2$ bits of entropy at a time, to ensure $a \leq s/2$, using Equation 3.14 each time to calculate new counter value [31].

Chapter 4

Data Collection

Data was collected from four real world random number generators with entropy input, the Linux random number generator, the FreeBSD generator, `prngd` as well as `HAVEGED`. The reason that these generators were chosen to gather data from was mainly due to availability. Only very few generators are implemented as user–space programs which included `prngd` and `HAVEGED`. The `EGD` generator also exists, but it functions similarly to `prngd`. Many operating systems do include generators that gather entropy which could be analysed. However, only Linux and FreeBSD were analysed as these operating systems were available and also had an open source implementation of the generators.

Reference generators were also implemented which provide random data following specific distributions. These generators are presented, together with details about how data collection was performed.

4.1 Reference Generators

Generators for different probability distributions were implemented to get reference entropies to compare the results to. These generators were implemented in python and got random numbers from `/dev/urandom`. Generated numbers were not IID and instead followed time varying distributions or were dependent variables having the Markov property. Output from the generators were numbers which were output to a file, with one number per line. Entropy for each generated number was also logged in order to have a reference entropy to compare with estimated entropies. Implemented generators where

IID choice

A generator consisting of multiple different IID distributions. Each individual IID distribution was over $\{0, 1\}$ with a distribution having chosen min–entropy. The min–entropies for the actual data were 0.2 0.2,0.3,0.5,0.5 and 1, where an entropy appearing twice meant two generators with that min–entropy were created.

For each provided min–entropy a distribution with chosen min-entropy was created. The generator did this by, for each min–entropy, randomly selecting either 0 or 1 to be more likely. The probability for the more likely value was the probability resulting in the desired min-entropy. Data was generated from one of the distributions while changing the currently active generator with a probability of $1/200000$

after generating a number. When changing generator, another random IID distribution among the available generators was chosen. Using this generator a total of $5 \cdot 10^6$ samples were generated.

Markov

A Markov generator where the distribution for each symbol following any given history was first randomized. The order of the generator used was 7, meaning that the distribution of each symbol depended on the 7 previous symbols. The possible output symbols of the generator were the integers 0 to 4. For each combination of 7 previously observed symbols the distribution of the upcoming symbol was randomized. This was done by assigning a weight at random, between 0 and 256 to each symbol. To provide more interesting distribution one symbol was randomly chosen to be more probable. This was implemented by randomly taking a number from 0 to 256 and added to the weight of this symbol. The probability of each symbol in this distribution was then simply the weight of this symbol divided by the total weight of all possible symbols. Using this generator a total of 10^6 samples were generated to be analysed.

Markov with State

Similar to the Markov generator with the same distribution but with the distribution occasionally changing. Each time a sample was generated, there was a one in 200000 chance that the distributions to use changed. This was done in a very simple way by simply having a number $s \geq 0$ and $s < 15$ corresponding to the state of the generator. Upon changing distribution this s was randomly selected among the possible values. The state impacted the distribution by multiplying the weight for output symbol 0 with s for all output distributions, leading to a noticeable impact on entropy. From this generator a total of $8 \cdot 10^6$ samples were generated to be analysed.

The actual entropy was calculated by taking the distribution for the next symbol and directly calculating the entropy for that symbol with the given history.

Time varying Binomial

A binomial distribution for the next random symbol with a constant number of experiments. The probability of success was altered linearly between two different values. This was done by assigning a constant weight of 100 to failure and letting the weight for success equal $|300 - s/1000 \mod 300|$ for sample s. The success probability was then simply the success weight divided by the total sum of the weights. This was then done with $n = 10$ experiments and the output number is the number of successful trials. Using this generator a total of 10^6 samples were generated to be analysed.

The entropy was calculated directly from the probability of k success trials. As this probability equals $\binom{n}{k}p^k(1-p)^{n-k}$ the entropy could then be calculated as a sum over all k.

Time varying Geometric

A geometric distribution where the success probability was altered linearly between two different values, in the same way as it was for the binomial distribution, when data was generated. The output from the generator was then, for each sample, the number of successful trials before the first failure was recorded. Using this generator a total of 10^6 samples were generated to be analysed.

To calculate the entropy of the geometric distribution where the success probability was p the formula

$$H = \frac{-p\log(p) - (1-p)\log(1-p)}{p}$$

was used.

4.2 Linux Random Number Generator

Interesting data to study in the Linux random number generator is the internal input to the generator, that is used to estimate entropy received. Furthermore, the data is not modified by functions that make the entropy harder, if not impossible to analyse. As this data is unavailable from outside the kernel, collection was done by modifying the Linux kernel to monitor the input to the `add_timer_randomness` function. This function adds entropy from timing of user– and disk–timings. Initially this was done through the `printk` logging function in the Linux kernel, causing data to be written to the system log. More functionality in the logging was eventually desired and the kernel was further modified to allow logging over network. This was done with the `netpoll` interface which allows sending of UDP messages from the kernel. These were collected on another computer that listened for data using `netcat`. Logging method to use was easily toggled by using module parameters, allowing output method to be changed while the kernel was running. By providing the kernel's entropy estimate in the log as well, it could be compared with external estimates.

Network logging was done as writing collected data to the system log could cause differences in system behaviour, potentially generating disk-events on its own. Sending collected data to another computer should generate less disk–events on the local computer. Some impact on the collection when using `netpoll` is possible as well, but impact should be limited to interrupts. Therefore, disk events should be logged without behaviour changes, except for a small runtime overhead.

Data related to added interrupt randomness was also collected. This was done similarly, but with logging performed on the `add_interrupt_randomness` function. These two functions are the only functions that add internal entropy to the input pool. Entropy is also added from hardware generators and entropy input to `/dev/random`. The two functions are however the only ways internal entropy is gathered.

4.3 FreeBSD Fortuna Generator

The FreeBSD operating system uses the Fortuna random number generator [40] and does therefore not need entropy estimates. The Fortuna implementation in FreeBSD gets entropy from varying events happening in the kernel, such as keyboard and mouse events, interrupts, and network traffic. In the actual implementation a queue system, where entropy is added to a queue of events, is used. Another thread then harvests entropy from the queue. The kernel also allows modules to define other RNG algorithms to be used with the entropy, with an implementation of the Yarrow algorithm always available. Algorithms that require entropy estimates receive this from the entropy sources, as they include an estimate when adding entropy [40].

Entropy collection was done with the DTrace tracing framework. This allows probing the system in several ways, such as tracing entry and return points for most kernel function calls, including functions handling entropy inputs. Logging was done by tracing calls to the function `random_harvest_queue` and recording arguments provided to it. Arguments were a pointer to provided entropy, size of provided entropy, estimated entropy content in data and a value related to origin of the data [41].

The function `random_fortuna_process_event` is also interesting as it actually uses the entropy. Before this function is called, the entropy is packed into a structure together with estimated entropy of the event, size of the event, origin of event and destination Fortuna pool. Furthermore, the structure contains a counter corresponding to the processor cycle count. The destination pool chosen for messages is cycled among the pools separately for all different sources. The structure that the entropy is packed into is 128 bits large with 64 of these bits dedicated to the input entropy. If the input is larger than 64 bits, the hash of the data is instead stored in the structure. The hash size is set to 32 bits and thus only half of the 64 bits of data contain entropy. To gather the cycle counters this function was also traced which allowed logging of hashed entropy and the cycle counter.

4.4 HAVEGED

Output of `HAVEGED` is not very interesting by itself as data has already passed through the inner PRNG of the program. Timings of the program are however interesting as these are the only entropy sources used. To get these timings, the `HARDCLOCKR` macro was modified. This macro is used to get the current time and was modified to also print the time. This logging could have some impact on data gathered as printing will modify the execution time. Impact on amount of entropy should not be large as outputting data should have a somewhat constant execution time that simply is added to all measurements.

4.5 Pseudo Random Number Generator Daemon (prngd)

To collect data from `prngd` the source code of the program was modified to make every addition of entropy in the function `rand_add` to be logged. The total estimate of added entropy according to `prngd` was also kept as a reference value. Programs which were called to gather entropy were defined in a configuration file which for the test was the example configuration file for Linux-2.

Chapter 5

Predictor Modifications

The idea of predictors to estimate the min-entropy of data sources seems to have been successful. However, there seems to be several changes which could potentially allow better entropy estimations. The approach of taking an upper bound on predictability and calculating min–entropy based on this could also potentially be changed to allow other entropy estimates. The modifications which actually were implemented and how they were implemented is described in Chapter 6.

5.1 Entropy Estimate

The original predictors use the fraction of correct predictions as basis for a min–entropy estimate for the whole source. This can be altered to instead give an estimate of Shannon entropy for each sample individually. Such an estimate is more useful when comparing estimates with a source with its own entropy estimates. A min–entropy estimate for such a source is of little use when the source produces predictable sequences. The predictors would give the whole source a low entropy even when only some sequences were predictable, which the source might already have accounted for in its own estimate.

To instead estimate Shannon entropy the predictor produce an estimated probability distribution $q_i(x_j)$ for symbol number i for each possible upcoming symbol x_j. This could be achieved with any procedure that estimate probabilities, such as Markov chains. The procedure that estimates probabilities may vary for different sources, as sources behave differently. For some sources Markov chains may provide good estimates while other sources may require other procedures to be implemented. The estimate of entropy for an added symbol is then $\hat{H}_i = -\log(q_i(x_j))$ which can be shown to give an overestimate of entropy in expectation.

Defining $p_i(x_j)$ for a symbol x_j from a random variable $\boldsymbol{X}_i$ as $p_i(x_j) = P(\boldsymbol{X}_i = x_j)$ gives that the expected estimated entropy for sample i becomes

$$E(\hat{H}_i) = -\sum_j p_i(x_j)\log q_i(x_j) = -\sum_j p_i(x_j)\left(\log\left(\frac{q_i(x_j)}{p_i(x_j)}\right) + \log(p_i(x_j))\right) = D(p_i||q_i) + H_i \tag{5.1}$$

where $D(p_i||q_i)$ is relative entropy between p_i and q_i as defined in Equation 2.13 and H_i is the actual entropy of the distribution $\boldsymbol{X}_i$ which the sample came from. As the relative entropy is positive this gives that

$$E(\hat{H}_i) \geq H_i \tag{5.2}$$

and thus the expected value of this estimate is greater than the actual entropy. This is thus an overestimate of the entropy in expectation, unless $p_i = q_i$ in which case the expected estimated entropy is the actual entropy.

By using the estimator one symbol at a time, only allowing it to base estimations on previously read symbols, will give an estimate of $H(X_i \mid X_j : j < i)$. Thus, this is an estimation of entropy rate of the source with knowledge of all previously generated samples.

Using this method, the estimate entropy of a sequence of samples corresponds to the logarithm of the probability of the sequence. For a sequence with h bits of min–entropy, the probability of estimating less than $k < h$ bits of entropy can be bounded. This is done by noting that the most probable sequence occur with probability 2^{-h}. Estimating a probability of at least 2^{-k} for a sequence can be done with the highest probability by estimating that probability for the 2^k most probable samples. In the worst case scenario, the probability of these sequences is actually 2^{-h}, giving that the probability of estimating k bits of entropy cannot be more than 2^{k-h}. This shows that this method of estimating entropy will underestimate the min–entropy by $u = h - k$ bits with probability less than 2^{-u}.

5.2 Predictor Collections

Predictor ensembles allow multiple predictors to be used in parallel without consistent underestimates of entropy due to several similar predictors. The implementation in the original paper [8] was by using the predictor with the global best prediction results. This is a reasonable approach if the source behave the same way for all data. If the random data instead comes from several distributions, a local score to determine which predictor to choose might be better. This would allow predictors to adapt if the type of output from the entropy source changes, allowing another predictor to take over.

Using the idea in Section 5.1 also allows more advanced ways of combining the predictors. One possibility is to give each predictor a weight, corresponding to how good it should perform. The ensemble probability can then be estimated by the weighted average of the probabilities which the predictors estimate for the sample. This could further be altered by dynamically assigning the weights, giving predictors which performs good larger weights. Dynamic weights allow the collection to take local changes into account. The chosen approach was to update the weights for a predictor to be the probability it estimated for the last W samples, for some chosen value of W.

Models contributing to the weighted average are only the models that give a probability estimate for the sample. In order to ensure that the probability for all possible different samples still sum up to 1, the models are required to not base their failure on the sample to estimate. They can thus only fail based on what the previously observed values are. The set $\boldsymbol{P}$ of all predictors P_k in the collection that give predictions with given history is then constructed. The estimated probability $\hat{p}$ of the collection is then calculated as shown in Equation 5.3. An example of the process when a collection of three predictors receive a sample x_i is shown in Figure 5.1. As $\boldsymbol{P}$ is independent of current sample x_i and $P_k(x_i \mid x_{i-1}, \ldots, x_1, x_0)$ is a probability distributions for all k, this normal-

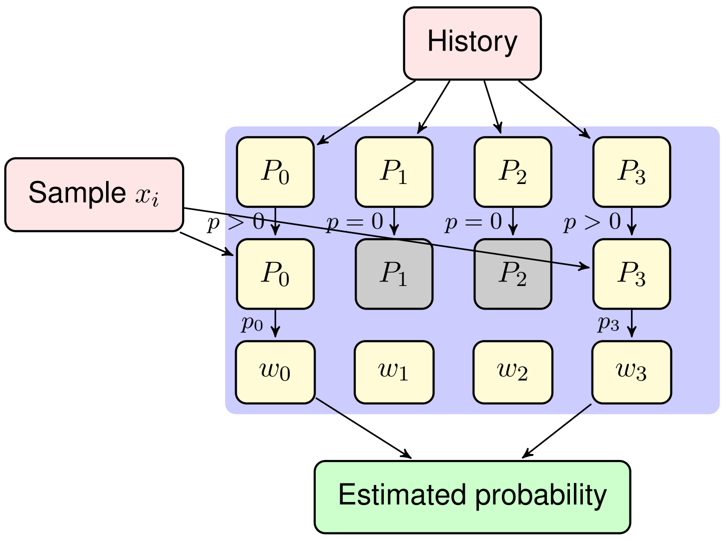

Figure 5.1: Diagram over how a collection with three predictors P_i works when receiving input

ized weighted average will also be a probability distribution.

$$\hat{p} = \frac{1}{\sum_{k\ :\ P_k \in P} w_k} \sum_{k\ :\ P_k \in P} w_k P_k(x_i \mid x_{i-1}, \ldots, x_1, x_0) \tag{5.3}$$

Using this approach it is not only possible to take some local differences in source behaviour into account, but it also allows use of estimators that require training data before making good estimates. These models will not perform well initially and may be unable to give probability estimates before more samples are seen. However, they will eventually receive enough training to potentially outperform other estimators and will then contribute to probability estimates.

5.3 Transforming Input

Some predictors in the original paper [8] were simple and did predictions based on single assumptions. These include the lag–predictor, the first order difference predictor as well as the MCW predictor. These predictors could potentially be combined with another predictor to produce better output. The original lag–predictor predicts the same value as seen n values ago. This works for the simplest types of periodic behaviours, but more advanced behaviours will go unnoticed. Another approach that potentially detects more advanced periodic behaviour is to feed the lagged stream into another predictor. A predictor of some other type then only receives every n:th input and is only meant to predict every n:th input. With in total n such predictors advanced periodic behaviour with period n could potentially be detected.

The first order difference predictor could also benefit from a similar change. The original predictor predicts that the next value is the previous one plus the difference between the previous value and the one before it. If instead using the difference as input to another predictor, it might be possible to detect more advanced behaviours that can better be described by differences between numbers. This could also be done with multiple layers of first difference predictors to get an n-th order difference predictor.

The most common in window estimator assumes that there is a distribution which changes with time, although not too quickly. A window of samples is therefore kept to make predictions on only those. The predictions are however based only on which sample is most common in the current window. A potential improvement could be to allow another predictor to make predictions based on the values in the window. This might not be as efficiently implementable as guessing the most common value, depending on choice of underlying predictor. Therefore, it might not be reasonable to slide the window at every sample. Another version that slides the window less frequently could however be feasible.

Doing these types of input transformations may allow the distribution to be changed so that it becomes easier to analyse. For example, a sequence of measured times has a first order difference whose distribution should be easier to analyse. Furthermore, it could be possible to transform some kinds of random data from non-independent data to IID random data. The random variable

$$\boldsymbol{X}_i = \boldsymbol{X}_{i-1} + \boldsymbol{Y}_i \tag{5.4}$$

for some random variable Y_i with Y_i and Y_j being IID for all i, j is a simple example. It is then easily seen that the first order difference is independent and identically distributed.

$$\boldsymbol{X}_i - \boldsymbol{X}_{i-1} = \boldsymbol{Y}_i \tag{5.5}$$

Chapter 6

Implementation

This chapter contain details about the construction of the entropy estimators. Implementation of the estimators was done in C++ while scripts that performed the tests were written in Python and Bash.

To allow the predictors to be used with several sources the implementation was done modularly. This allows easy configuration of predictors to use by simply editing a configuration file. As the predictor can be modified to match the source, this could allow good estimates for different types of sources.

6.1 Data Format

In order to allow the method to easily be used on different types of sources the implementation was easily configured to use different versions of data formats. Internally the data was represented with the help of templates in C++. These templates could be used with different types, allowing the same method to be used on different types of data. The available data types were long, char, double, string and bits. For the long and double types data was read through standard C++ input. In practice this meant that this type of data was read with one entry per line of the file. For the other types of input, bits and char, data was instead read from the binary representation of the input file. For the char input type, each sample was simply 8 bits of data read sequentially from the binary representation of the file. The "bits" data type was similar, but instead of always having 8 bits per sample the number of bits per sample was configurable.

6.2 Estimations

Initially the entropy estimations were based on the same approach as in [8]. However, it quickly became clear that this approach would not produce very interesting result when used to analyse the entropy in the Linux RNG. The reason for this was that the distribution of the difference between the timings was heavily favoured towards the output 0. As such a large portion of the differences was identically equal to zero, the best prediction almost always being equal to 0. As such, the prediction success rate became almost equal to the portion of time differences which were equal to 0. This number was not very interesting as the kernel estimator already estimates the entropy added by all these 0 differences samples as being 0. Transformations of the data could potentially fix this problem, but another approach was chosen instead.

The approach was the same as presented in Section 5.1, where probability of upcoming samples was estimated. Estimating sample probabilities may be a harder problem than guessing which sample is next. However, some models can be adapted to estimate probability instead of predicting the next sample. This includes Markov chains which was the underlying model that was mainly used in this thesis.

6.3 Markov Chains

The estimator used to actually produce the estimated probabilities was Markov chains of varying order. A Markov Model of order L was constructed using a C++ object of type `unordered_map` with keys being arrays of the L previous elements while the values were other `unordered_map`s. These maps contained data on samples following this history. The keys of the maps were the samples which had occurred while the values were the number of times they were seen. To implement this a hashing function for arrays was required. This was implemented with the help of the `boost` [42] library and the function `hash_range`.

The probability of a new sample for a given history of L samples was estimated as the fraction of samples which followed this history that were equal to the given sample. To avoid estimating zero probability for samples not previously observed with the given history, all unobserved samples were calculated as one extra data sample when determining the probability. To calculate probabilities correctly it was then necessary to know all possible samples before running the test. Predictors were thus allowed to look at all samples which could be fed into them, before the test. The predictor could then estimate the probability of known samples as usual, but with the count of samples following any given history incremented by 1. The probability of a previously unseen sample was then based upon the number U of samples which still had not been seen. With a count of C previously seen samples after this history the probability was set to be $p_u = ((C+1) \cdot U)^{-1}$. This ensures that the total probability of all samples is always 1 as the sum of the probabilities of the seen samples is $C/(C+1)$ and the probability of the unseen events is $U \cdot p_u = 1/(C+1)$.

6.4 Collections

The collection object allowed multiple different predictors to be used together. This was done by letting the collection have an array of other predictors which each make their own independent probability estimates. The collection gives the contained predictors all data it reads and estimates the probability of a sample as a weighted average of probabilities from all predictors giving non–zero probabilities. The weights are the normalized estimated probabilities for the W previous samples for each predictor. To actually keep track of this probability, the collection keeps a queue of negative logarithms of estimated probabilities for previous samples for each predictor it contains. This allows a running sum of entropy estimates which the first element in the queue is subtracted from when the queue contains at least W samples. Logarithm laws give that the estimated entropy of the previous samples directly correspond to the probability of these samples, which gives a way to determine the weights. The probabilities are however not calculated as for large values of W the probability of any specific long sequence of data almost always is very low. This may lead to rounding problems during calculations and probabilities may be so small that they are represented as 0 during calculations. Instead, estimated entropies of the samples were used to calculate the weighted probabilities. With k contained predictors, were predictor i have previous entropy H_i and thus probability 2^{-H_i}, the normalized weights w_i are calculated as in Equation 6.1. This removes problems related to small numbers as $2^{H_i-H_j}$ will be sufficiently large for all j that give non–negligible impact to the weighting.

$$w_i = \frac{2^{-H_i}}{\sum\limits_{j=0}^{k} 2^{-H_j}} = \frac{1}{1 + \sum\limits_{i \neq j} 2^{H_i - H_j}} \tag{6.1}$$

6.5 Delta Transformation

The delta transformation was created to better estimate entropy in sources where the unpredictability mostly was in difference between different observations. As such it was mostly relevant for data which were logically represented as numbers. This transform contained another predictor which made the actual predictions while the delta predictor only keeps track of previously observed values. When there is a previously observed value, it gives the contained predictor the difference between the new sample and the previous sample as input data. It also sets the previously observed to the currently observed one. Estimated probability of a sample is the contained predictors estimated probability of the difference between the given sample and the previously observed sample. The difference function is invertible if given the initial value meaning that the entropy of the samples equals the entropy of the differences plus the entropy of the first sample. When several thousand samples are analysed, the entropy of the first sample can however be considered negligible and be ignored. The transformation can in many cases be replaced by a simple program calculating the difference of data ahead of time. The transformation does however allow increased flexibility by combining it with other transforms and decrease the need for data preparation.

6.6 Other Implementations

More transformations of predictors were constructed but were not used for the results in this report. These transformations may be useful for different types of entropy sources and are briefly presented here. These were easily available for use by changing a configuration which was read with the `libconfig` library[43].

6.6.1 Lag Transformation

A transformation with arbitrary long lag n was implemented as a predictor transformation. This was done by requiring and array of n predictors and alternating between them when constructing the predictions. In this way, predictor i was estimating probabilities for samples $kn + i$ for all values of k. This was also the only predictor which received this sample as input data.

6.6.2 Moving Window Transformation

A transformation which only allowed a predictor to be trained on a moving window was also constructed. This transformation took a predictor, a window size w and a reset interval r as arguments. The transformation also required that the predictors were resettable, allowing removal of any trained data. The transformation then kept track of up to w previous observations while also feeding them into the contained predictor. Contained predictor estimated probabilities as normal until the window was full. Then the predictor was completely reset and the window was moved one reset interval forward. The predictor was then fed all data remaining in the window after it was moved. This results in the predictor, after the first fill of the window, always having between $w - r$ and w samples of history.

6.6.3 Function Transformation

A generalization of functions that could transform the input to the predictors was created with the function transformation. This transformation took a child-predictor and a function as argument. The function was fed a chosen number of previously observed samples and the current sample. Its output was then fed into the child predictor. The child predictor is thus only trained on transformed data and will estimate probabilities of transformed data. One example of a function transformation available in configuration files is the sum transformation. This transformation sums the n arguments received and feeds to the child predictor. This transformation can be inverted with knowledge of the first n samples. If entropy of the first n is small in comparison to total entropy, this estimate could be used as an approximation of entropy even without knowledge of first n samples.

6.6.4 Choice Transformation

A generalization which could implement most variants of lag–like transformations was implemented in the choice transformation. This took an array of predictors as input, together with a function. The function takes an array of previously observed values and the number of the current sample as input. It uses this information to decide which predictor in the array to use, causing that predictor to decide the probability distribution for the next sample. The function is also used to determine which predictor observed data is added to. No example of this is currently available as a transformation in the configuration file.

Chapter 7

Results

Results received through analysis of generators and entropy estimations are presented here. Entropy estimates from the predictors on data from the reference generators are also presented. To acquire per sample estimates of entropy the implementation described in chapter 6 uses the method described in Section 5.1. This allows an estimate, with high variance, of the entropy for each sample. This estimate could be plotted as a function of sample. However, to reduce the variance and produce better figures, the moving averages of these estimates as functions of samples are instead used for most plots.

7.1 Reference Generators

The entropy estimation for the reference generators was performed using a predictor consisting of a collection of Markov chains with orders 0 to 10. The estimated entropy is compared to actual entropy and plotted. The result for the choice IID with moving average over 20000 samples is shown in Figure 7.1. Results for the time varying binomial function are plotted with a moving average over 1000 samples in Figure 7.2. The results for the time varying geometric generator are shown in the same way in Figure 7.3. Results from the Markov generator with the same moving average over entropies is shown in Figure 7.5. These results came from a collection predictor and can be compared with the results shown in Figures 7.6 and 7.7 where only individual predictors are used.

Results from the state Markov model generator are plotted with a moving average with a window size of 10000 and are shown in Figure 7.4.

A simple comparison test was also performed between the modified predictors and the original implementation. They were both tasked with entropy estimations on uniform distributions for different entropies. This gave the graph shown in Figure 7.8 where the modified predictors were run with collection of Markov predictors with order 0 to 10 and window size of 10. Actual run time comparisons were not performed, but the majority of the time was spent by the original predictors during the test.

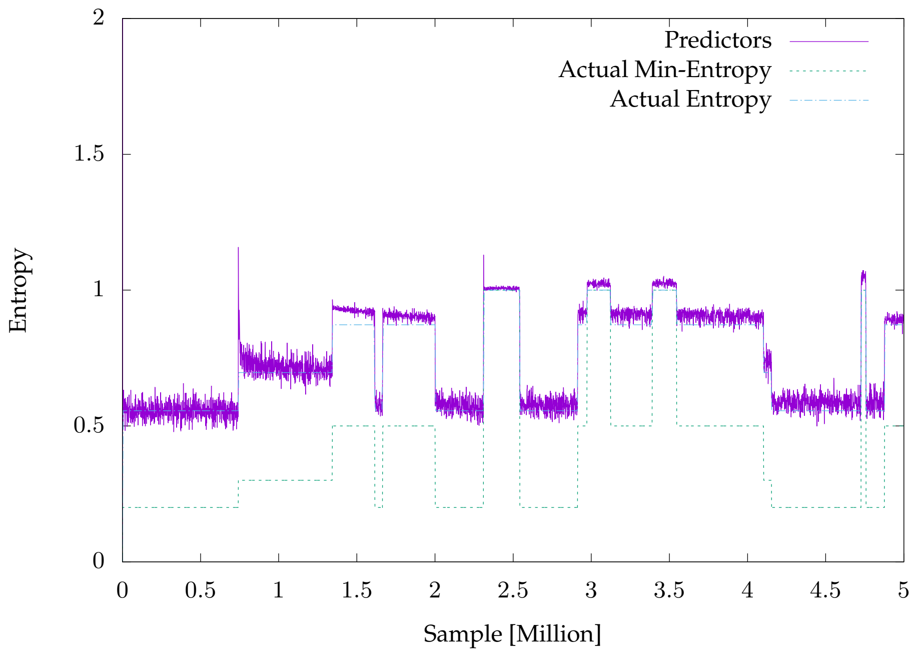

Figure 7.1: Estimated and actual entropy for Choice IID generator

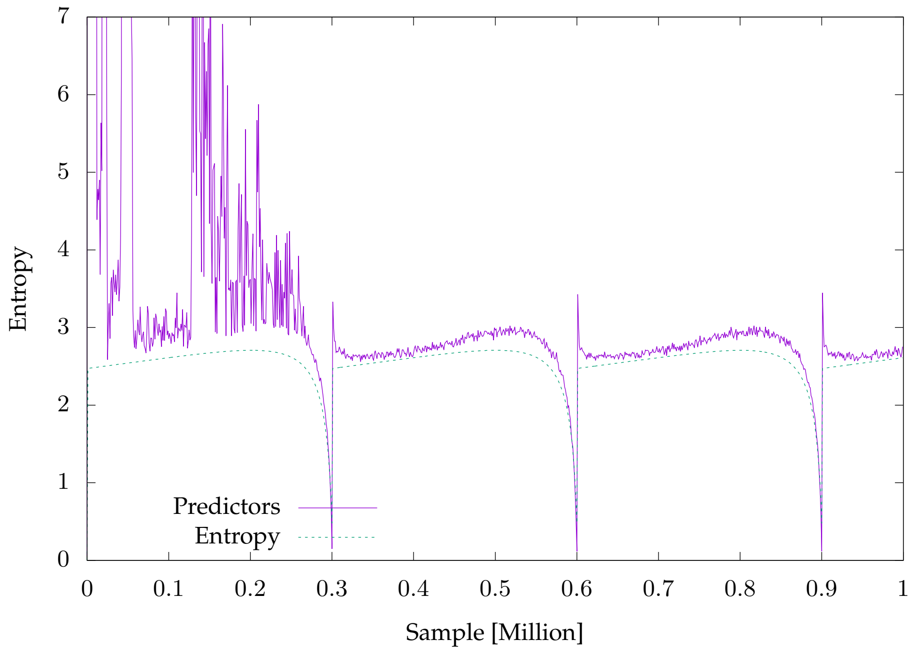

Figure 7.2: Estimated and actual entropy for time varying binomial generator

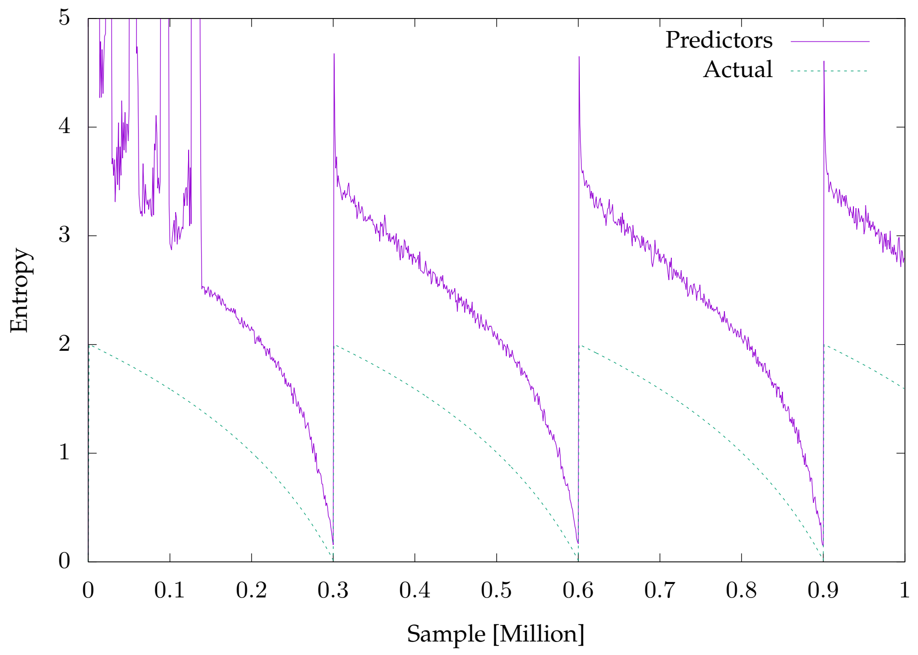

Figure 7.3: Estimated and actual entropy for time varying geometric generator

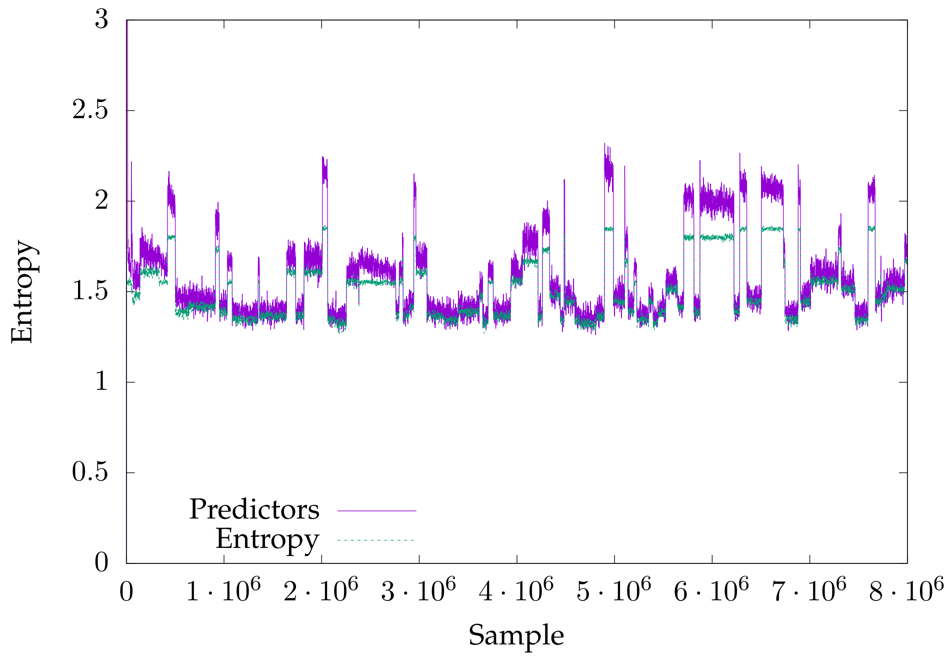

Figure 7.4: Estimated and actual entropy for Markov generator with states

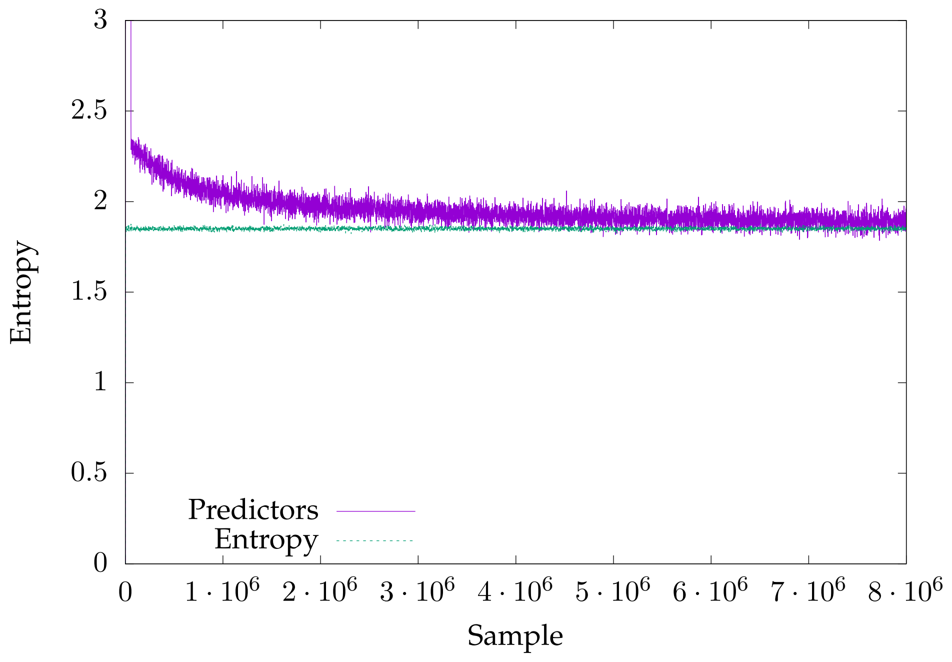

Figure 7.5: Estimated and actual entropy for Markov generator with collection predictor with order 0 to 10 and window size 1000.

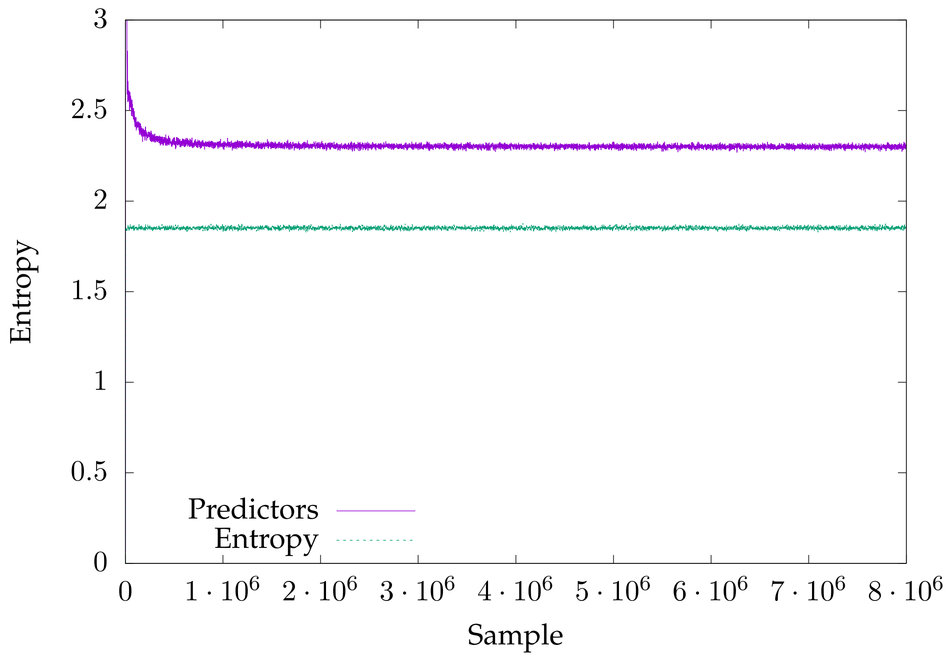

Figure 7.6: Markov predictor with order 5.

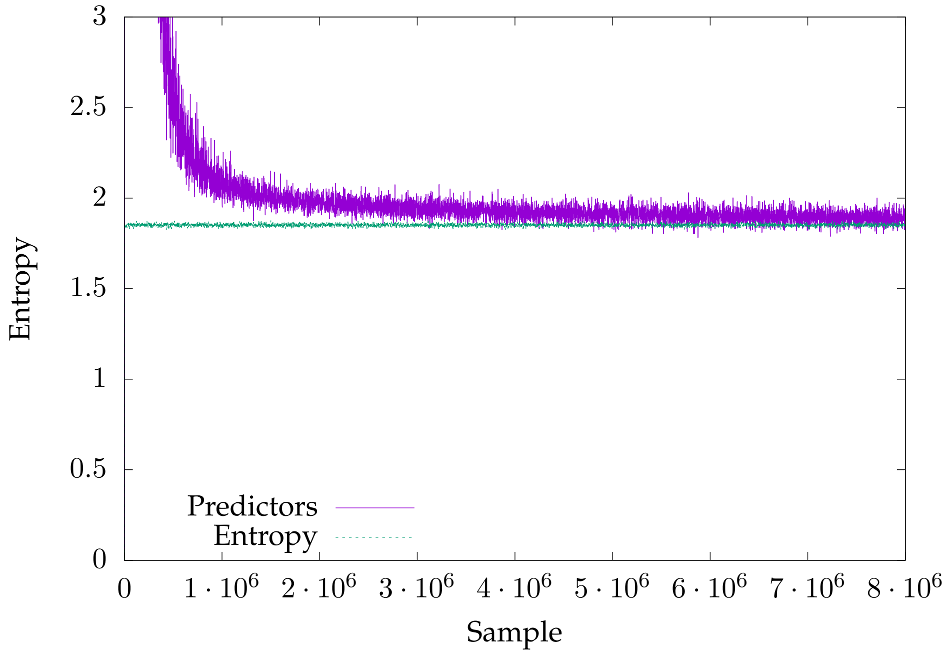

Figure 7.7: Markov predictor with order 7.

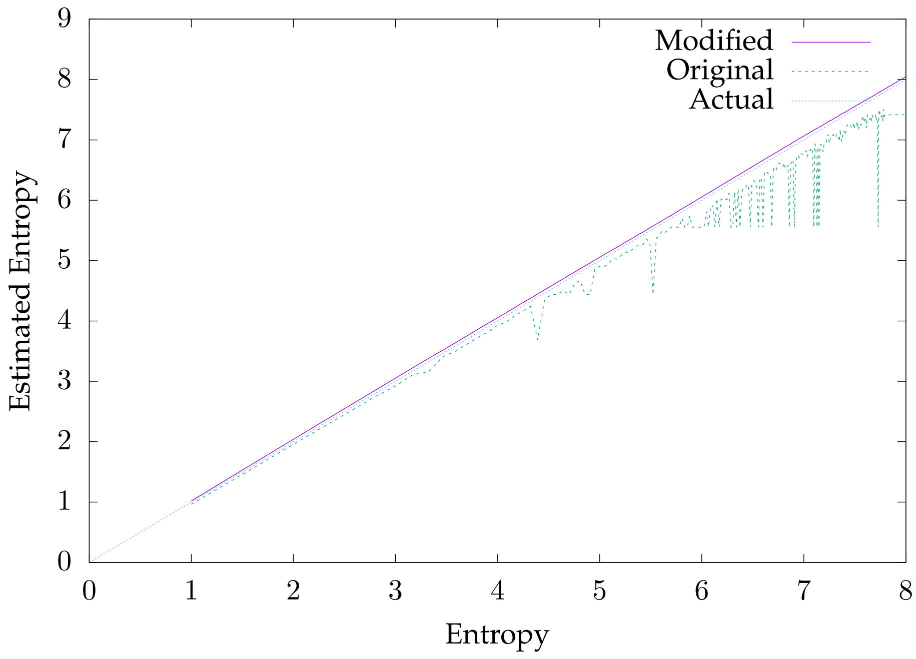

Figure 7.8: Comparison between modified and original predictors on uniform distributions of different entropies. For each entropy in the graph, a total of 50000 samples were analysed for each method.

7.2 Linux Entropy Estimation

The gathered data gives a large sample set to estimate entropy on. Using predictors with the modifications previously described with a collection with Markov chains of order 0 to 11 gives the result shown in Figure 7.9. The analysed data is the time of the event in jiffies, which is the same data that is used for entropy estimation in the kernel. The estimated entropy varies heavily between samples, causing the estimation for any sample by itself to not be very interesting. As such, the moving average of estimated entropies is calculated and plotted instead.

As there is many recorded samples, details can not be visualized directly in a plot. A couple of other measurements were instead calculated for each test. These measures were related to the difference between the estimated entropy by the kernel and the one estimated by the predictors. As the predictors should provide overestimates of entropy while the Kernel attempts to provide underestimates, the interesting cases are when predictors estimate a lower entropy than the kernel. When this is the case, the kernel will have overestimated the entropy, which may lead to a security vulnerability. The largest such overestimate performed by the Linux kernel in comparison to the predictors was one such measure. Another measured quantity was the lowest entropy that the predictors estimated for a sequence which according to the kernel contained at least x bits of entropy. Here x was some constant, for example 256 bits. This was represented by a factor f with the predictor estimate being $x \cdot f$.

The device from which the entropy was recorded was a Lenovo ThinkPad E540 running Arch Linux with kernel version $4.9.6 - 1$. The only kernel modifications were those used to log data with `printk` as well as `netpoll`. Data was collected during multiple sessions under different conditions. The conditions for the different sessions are presented below.

- 1-9,11 Logging with `printk` with normal usage.
- 10 Logging with `printk` with computer left alone for longer periods of time with little to no interaction from outside.
- 12 Logging with `netpoll` with computer left alone for longer periods of time with little to no interaction. Interrupts logged as well.
- 13 Logging with `netpoll` while computer was left alone for ≈ 10 hours.
- 14 Normal usage with `printk` logging and interrupts recorded.
- 15 Computer left turned on over weekend with `netpoll` logging. Periods of usage and other periods where it was left alone.

Another measure recorded was the fraction of the recorded events which came from interrupts and the fraction coming from timer randomness. It turned out that this varied heavily between different sessions. In sessions 12 and 15 it was approximately twice as many timing events as interrupt events recorded. For session 13 there was 23 times more interrupt data than there was timing data. Finally, for session 14 there was ≈ 13.5 times more timing data than interrupt data.

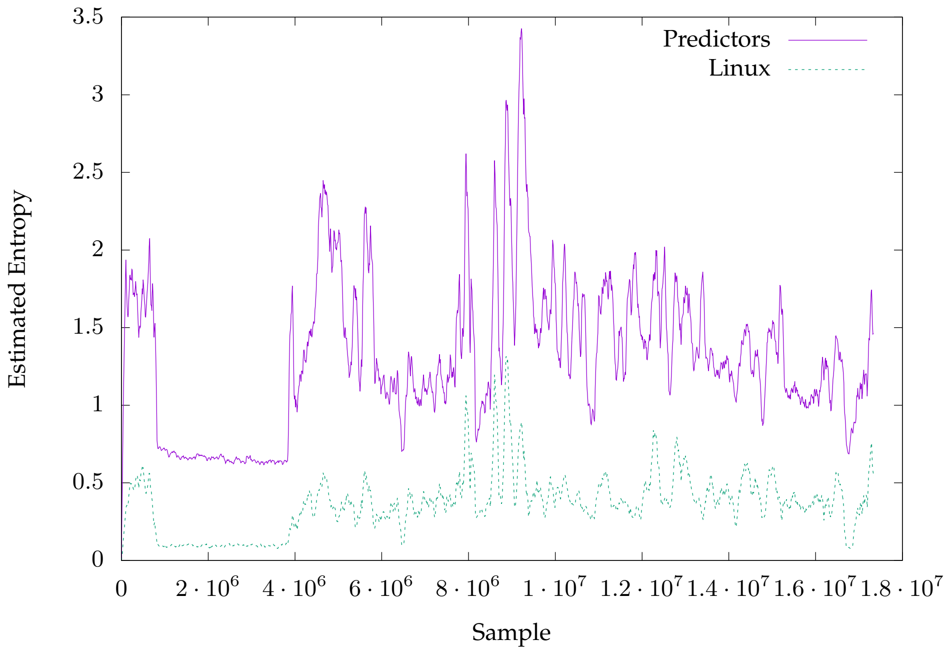

Figure 7.9: Moving average of estimated entropy with window size of 100000.

7.2.1 Timer Randomness

The results in Figure 7.9 show that the predictors have a higher moving average of estimated entropy with this configuration. As the window size for the moving average is large, this figure does not show local behaviour. With smaller window sizes the results have too high variance to be represented in one big figure. Instead, tables with the measurements introduced in Section 7.2 are presented.

In Table 7.1 the result when using a predictor with a first order difference transform containing a collection with Markov predictors are shown. The Max column shows the maximal found overestimate of the Linux kernel in each session. This is the maximal value that the kernel estimate is above the predictor estimate for any sequence of samples. The factor column shows the minimal factor f as defined in Section 7.2 with $x = 256$. Finally, the last two results show the average estimated entropies over the whole session for the predictors as well as for the Linux kernel.

For results which do not use the collection construct see instead Table 7.2 where the best performing individual Markov chain predictor is presented for every measure. The best performing predictor varied between sessions and measure and thus each measure also have a "From" column which specifies which Markov chain order the results are from.

To summarize the fraction measure with $x = 128$ the proportion of sequences estimated to have a factor f as a function of this factor were computed. The results of this from some sessions are shown in Figure 7.10.

7.2.2 Interrupt Randomness

Interrupt data was gathered in four different sessions, during which the computer saw varied use. Three of these sessions were logged over the network via `netpoll` while the

no	Max	W	Factor	W	Average Predictors	W	Average Linux
1	134.674395	100	0.581942	10	1.55514	1	0.46777
2	124.898647	1000	0.920068	1	1.86665	10	0.47906
3	86.5105118	200	1	-	1.51781	1	0.41592
4	121.718951	100	1	-	1.42132	1	0.40999
5	69.114083	100	1	-	1.3788	1	0.35713
6	418.888374	10	0.219050	50	1.6197	10	0.45663
7	496.075764	10	0.344601	10	1.49834	1	0.50305
8	439.926523	30	0.208616	10	1.29448	1	0.43889
9	447.971451	30	0.110233	1000	1.16078	1	0.36245
10	37.2460803	30	1	-	0.659372	10	0.09980
11	561.679079	10	0.158648	200	1.53241	1	0.35324
12	18.8769	10	1	-	1.77136	1	0.50184
13	20.30134	200	1	-	2.44366	1	1.13492
14	721.866697	10	0.102529	1000	1.10618	1	0.32868
15	586.769148	10	0.099985	1000	1.53631	1	0.63045

Table 7.1: Results for data from multiple recording sessions with predictor consisting of a collection with Markov predictors of order 0-11. The window size for the collection is the size which gives the best result among 1,10,30,50,100,200,1000 and is, for each result, displayed in column W

no	Overestimate	From	Factor	From	Total	From
1	27.346235	2	1	-	1.99133	0
2	31.672468	4	1	-	2.15932	0
3	12.63279	0	1	-	1.92175	0
4	11.673814	1	1	-	1.81277	0
5	10.288552	0	1	-	1.72979	0
6	87.623139	2	0.72305	2	1.98867	0
7	178.29783	2	0.67548	2	1.902	0
8	195.18017	2	0.33213	2	1.68046	0
9	335.12425	2	0.27496	4	1.507	0
10	15.0	6	1	-	0.72669	0
11	219.21054	2	0.50294	2	1.81151	0
12	18.0	4	1	-	2.07246	0
13	15.852008	0	1	-	2.60901	0
14	293.23478	2	0.27729	4	1.4107	0
15	45.255055	2	0.85787	2	2.04174	0

Table 7.2: The maximal overestimate, minimal factors and minimal total for Markov predictors of only one order. From columns specify which order gave the best value.

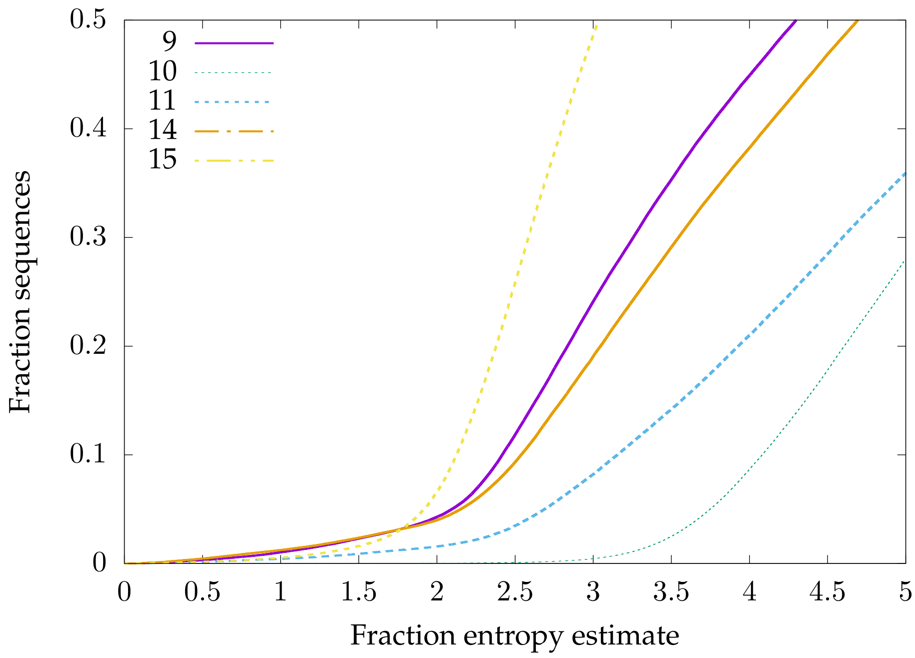

Figure 7.10: Proportion of sequences under fractions f plotted as function of f with $x = 128$ bits

last one was logged locally with `printk`. Estimated entropy for interrupt codes and interrupt timings for the longest of these runs are shown in Figures 7.11 and 7.12. It should also be noted that only the time in jiffies was analysed and higher precision timers which added entropy were ignored. This session, number 15, was performed over a weekend where the laptop used for the experiment was left on, constantly sending events to another computer. The laptop periodically saw some usage during this time but the majority of the time it was left more or less alone. The plots show the moving averages over 5000 entries with one of every 1000 samples shown in the plot. Data from the other sessions with recorded interrupts recorded behaved similarly.

It was also seen that one of the interrupt codes that provides entropy comes from the `i8042_interrupt` which is related to keyboard input. When this interrupt is performed, a keyboard specific routine is called. The keyboard data is then interpreted and passed along as user input. This results in the method `input_handle_event` to be called multiple times, each time adding data as input randomness.

7.3 FreeBSD Entropy Estimation

Data collected from tracing with DTrace was content of memory segments with size 136. This data was transformed by translating each sample to a number indicating the number of other unique samples seen before this sample was first seen. If the translation table is considered known then this will not impact the entropy estimate as with it the translation is a one–to–one function. As the translation of a sample will be known after it is first observed, entropy will only change for the first observations of a sample, even without translation table knowledge. If enough samples are observed, the impact of the first observation of a sample can be considered negligible and thus the translation can be

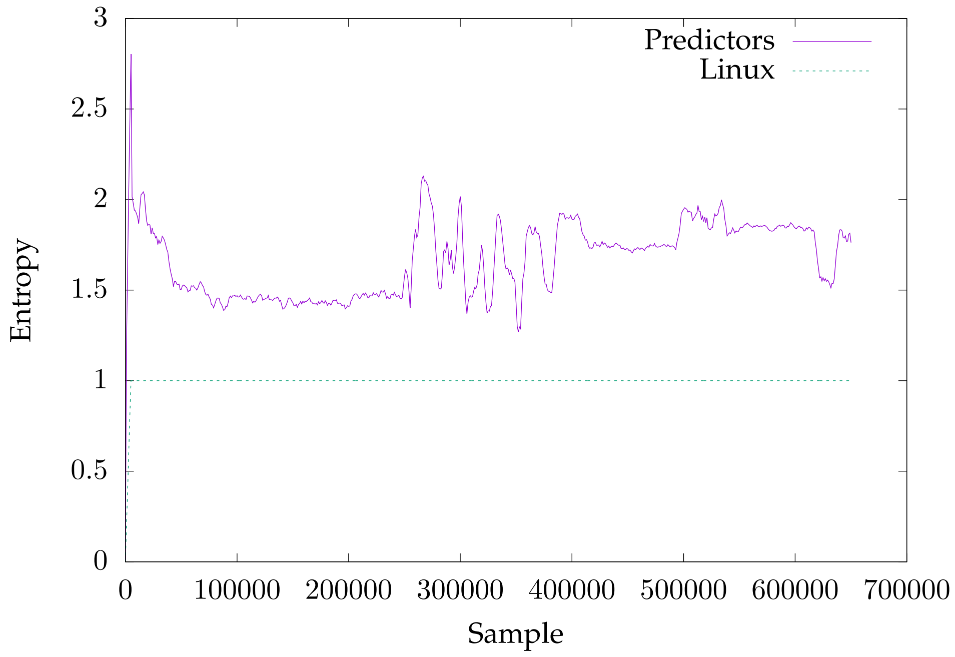

Figure 7.11: Moving average of size 5000 over estimated entropy for interrupt codes

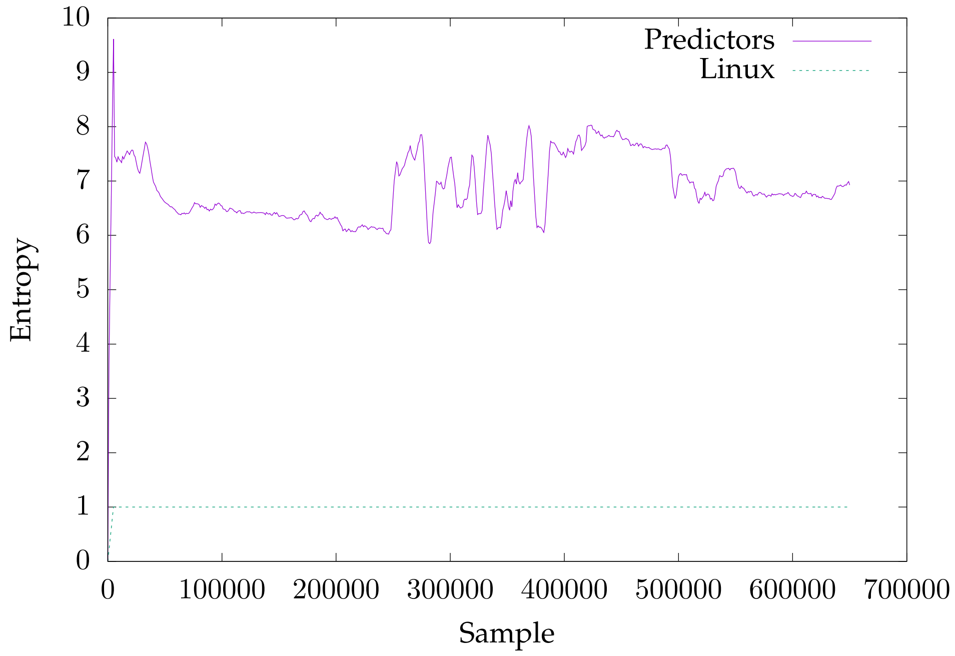

Figure 7.12: Moving average of size 5000 over estimated entropy for interrupt times

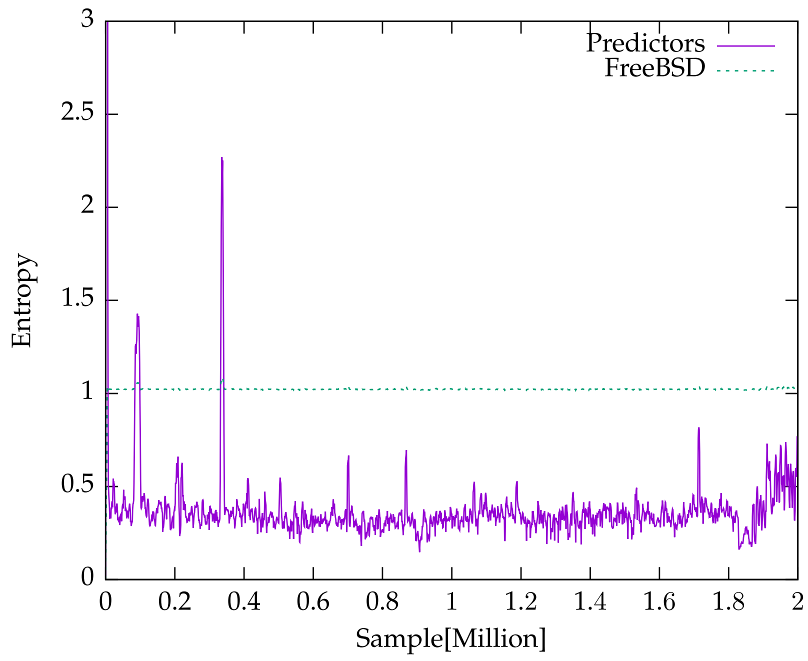

Figure 7.13: Estimated entropy in data with estimations by predictors and FreeBSD

used with only negligible entropy impact.

Output of this translation was analysed with the modified predictors which estimate the entropy of the sequence of samples. This estimate was then combined with the estimate available in the kernel. The moving average of these values with a window size of 5000 was plotted with results shown in Figure 7.13.

7.4 PRNGD Analysis

The configuration file used contained several programs which were called to get entropy the same program were used multiple times with different arguments. Programs in the example configuration file were, `ls`, `netstat`, `arp`, `ifconfig`, `ps`, `w`, `who`, `last`, `lastlog`, `df`, `vmstat`, `uptime`, `ipcs` and `tail`. Results of the commands vary depending on setup and some commands may be unable to produce unpredictable output on some configurations. The `ls` command is used on certain folders which it deems to often vary, such as `/proc` and `/tmp`. The `tail` command is used on different log files and hopes to exploit unpredictability in recent entries to the system log.

In practice, output from several of these programs is more or less constant on some machines. For example, output from `ifconfig` and `arp` have little changes while the same network configuration is used. Programs such as `who`, `last` and `lastlog` are more or less constant and predictable on single user systems. Furthermore, output from some programs depend on each other, with output from one program known causing output of other programs to be more predictable. Estimated entropies for the programs are all low in the used configuration, with between 0.01 and 0.05 estimated bits of entropy per byte output from the program.

7.5 HAVEGED Analysis

Results of analysing the HAVEGED algorithm and entropy estimations of its gathered entropy are presented here. Actual code for the HAVEGED generator is presented in Appendix B.

7.5.1 Algorithm Analysis

The HAVEGED generator works with the `oneiteration.h` file which is included inside another function. This function ensures that the contents of `oneiteration.h` is executed the necessary amount of times, while the actual random generation is defined in the `oneiteration.h` file. The array `PWALK` and the pointers `PT` and `PT2` are used for the random walks. Another variable called `ANDPT` is also used in order to ensure that the random walks are within the bounds of the array. Size of the `PWALK` array and therefore the value of `ANDPT` also depends on the computer cache size. On the machine performing the tests the `ANDPT` variable was $0x3fff = 16383$ corresponding to a size of the `PWALK` array equal to $0x4000 = 16384$. Output data is put into the `RESULT` array while a variable `i` keeps track of the position in the array. The `HARDCLOCKR` macro reads the current time and sets the provided variable to this time in nanoseconds. Using this, the first 8 numbers for one iteration are generated as shown in Figure B.1. While the numbers are generated, another variable `PTTest` undergoes several tests. These are meant to exercise the branch prediction and thus alter the running time of the program. The result of this variable is also used to potentially alter the output as shown in Figure B.2.

The next 8 outputs are generated as shown in Figure B.3 while the positions of the random walks are updated as well. At the same time more tests of the same type as in Figure B.2 are performed in order to alter the running time. These tests do however not affect the output directly and only impact the running time of the program and are therefore not included here. The initial state of the generator is with the contents of the `PWALK` array and the `PT` and `PT2` variables all equal to 0. Before any output is generated, the generator thus have to first be initialized. This is done by first running the collection loop 33 times which corresponds to fully gathering the `RESULT` array 33 times. The size of this array may be configured but the standard value is $128 * 1024 = 131072$. As each iteration results in 16 output values while reading the clock 2 times this means that the warm–up procedures reads the clock at least $128 \cdot 1024 \cdot 2 \cdot 33/16 = 540672$ times. Depending on execution behaviour the number of gathered values may be higher with up to 40 extra iterations after the array is full. Furthermore, if "online tests" are enabled more initial gathering runs are executed. During the tests they were enabled, resulting in a total of 61 fills of the result array before data was generated and thus a total of at least $61 \cdot 16384 = 999424$ times were used before the generator started giving output.

Postilions in the random walk determine numbers to alter and to output as results. These positions are selected via contents of the walk and attempts to be randomly chosen within the bounds of the array. Numbers chosen from a position P can be seen to be contents at positions P xor k for integers $0 \leq k < 8$. This means that the chosen numbers are a block of 8 positions around P with the last three bits of P only determining the order these numbers are chosen.

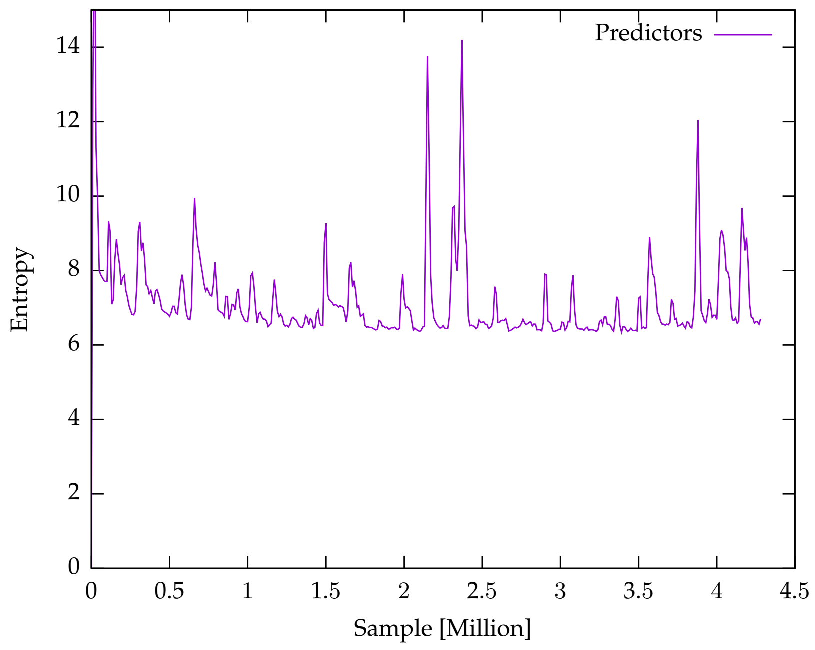

Figure 7.14: Estimated entropy for timings from HAVEGED

7.5.2 HAVEGED Entropy

Timings gathered from the HAVEGED generator were analysed to estimate the amount of entropy gathered by the program. The timings from the program when a total of 100 megabytes of data was generated to output file `/dev/null` was collected. The entropy of this was then estimated with the help of a predictor consisting of a difference transformation containing a collection of Markov chains with orders 0 to 10. A moving average over 20000 samples, with one sample for every 10000 samples plotted is shown in Figure 7.14. Over all samples the total average estimated entropy was 7.18065.

Chapter 8

Discussion and Conclusions

In this chapter, the results and their consequences are discussed. Conclusions about the performance of the modified predictors are also included and their results on the reference generators are discussed. A final conclusion about the state of random number generators with input is also presented.

8.1 Reference Generators

The results from the reference generators show that the modified predictors produce estimates which, as expected, overestimate the Shannon–entropy and. The results by themself do however not say anything about the predictors general performance at estimating entropy but they show how the predictors estimate for the given sources. The graphs show that even without any moving window predictor transformation the entropy estimation follows changes in actual entropy relatively well.

Comparing the plots in Figures 7.5, 7.6 and 7.7 show what type of impact the collection predictor has. In Figure 7.7 it can be seen that the Markov chain with the same order as the generator does converge towards the actual entropy. The collection generator also does this but with a difference in behaviour before convergence. Estimates made by the collection are lower than the estimates of the single order Markov predictor until about $2 \cdot 10^6$ samples have been analysed. After this, both predictors are mostly converged and only slowly decrease a bit closer to the actual entropy. It can also be seen in Figure 7.6 what happens when a too low order Markov Chain is used. Although the estimated entropy is lower than the correct order Markov predictor initially, it fails to converge to the actual entropy. Instead, it seems to converge to another entropy which is higher than the actual entropy of the generated data.

It can also be noted that the predictors react to changes in actual entropy quickly, even when the estimate is relatively far from the actual entropy. This can be seen in Figure 7.4 where the estimate is relatively far away from the actual entropy in some states. The estimated entropy does however still change quickly when the state changes and the actual entropy changes. Furthermore, the estimation seems relatively stable when the actual entropy for the underlying Markov Chain is more or less constant. For Markov processes this might be used to detect when there are changes in the underlying process. By observing the output of the predictor, stable and slowly decreasing segments of data may be identified with there being a constant state in underlying process. Similarly, sudden changes in the estimation may be identified with actual behaviour changes in the underlying process. The degree of correspondence between behaviours of predictors and actual process may however vary greatly between types of process. As such, a more through analysis would be necessary to find when this kind of correspondence exists and can be used.

8.2 Linux Estimator

Some discussion about the results from the entropy gathered in both the timings and the interrupts in Linux are presented here. This is followed by a discussion about what possible impact the results could have.

8.2.1 Timings

The entropy estimate performed in the Linux kernel, as shown in Formula 3.11, is based upon the logarithm of a parameter d depending on differences in timings. This correspond to estimating that the probability of the parameter is evenly distributed up to the current value, if rounding effects are ignored. This means that the estimator estimates that d follows a uniform distribution and estimates entropy from this. The estimate from the Linux estimator is thus an estimate of both the entropy and min-entropy of the symbol as the min-entropy equals the entropy for uniform distributions. Furthermore, the modified predictors estimations of entropy will only be lower than the actual min-entropy with a low probability. This means that if the predictors estimate a much lower entropy than the generator then there probably is something wrong with the generator estimate, as it should underestimate entropy. As seen in Table 7.1 such an incorrect estimate of entropy exists in all sessions. Smaller incorrect differences of less than about 20 bits of entropy can be attributed to chance. The probability of such a difference is $> 2^{-20} \approx 10^{-6}$ even with correct entropy estimate. With several million samples analysed, events which happen approximately once in a million are relatively likely to occur. Other sessions have larger overestimates by the Linux generator, which can not be attributed to chance. These estimates, which are between 37 and 722 bits too big, do show that the Linux estimation is not always an underestimate of entropy added with each sample.

Looking at data where the kernel overestimated entropy showed a repeating pattern. This pattern was from input randomness and consisted of repeating sequences of two different event codes. The codes are alternated, with the second code always occurring at the same time, in jiffies, as the first. Actual codes used varies between the different sessions but within each pattern, only two codes are used. For all observed such sequences the timings were very predictable with the time between pairs of events almost always being equal to 9 jiffies with some rare occurrences of an 8 or 10 jiffy difference. The predictors detected this pattern and gave high probabilities for these sequences somewhat quickly. The internal Linux estimator was however unable to detect such a pattern as δ varies between 0 and 9 meaning δ_2 alternates between ± 9 and δ_3 will be ± 18. As such, the kernel estimate of entropy in these events alternates between 0 and $\lfloor \log(9) \rfloor = 3$. However, entropy in the repeating pattern is much smaller as the outcomes are very predictable.

As seen in Figure 7.9 the total averages for the kernel estimate and the predictors estimate are quite far apart. Therefore, the kernel estimate seems to underestimate the actual entropy in most cases, although sometimes overestimating the entropy in the data. This shows that the estimation fails at certain types of input which the generator do receive during normal usage. The results do however also show that during larger time periods, the kernel estimate behaves similarly to the predictor estimate. The predictor and kernel estimates both follow the same pattern with high and low values occurring at the same time for both estimates. It almost seems like the predictor estimate simply is an upscaled version of the Linux estimate. The Linux estimate thus seems to capture most of the same kind of behaviour as the collection of multiple orders of Markov chains. As such, the Linux estimate does seem to efficiently make good estimates, if ignoring some shorter periods of overestimates.

The maximal values for overestimates and minimal factors presented in Table 7.2 are worse for all individual Markov Chains than the values for the collection. This shows that combining predictors into one single predictor do improve the performance of the predictors, allowing a combination that performs better than any individual predictor. Results from the single order Markov Chains do however also show that even single order Markov chains detect entropy overestimates in the collected data.

8.2.2 Interrupts

The entropy contained in the interrupts seems to be larger than what the kernel estimates. The predictors' entropy estimate for timings was higher than what the kernel estimated the contained entropy to. Analysing only the interrupt codes, the predictors still were unable to produce a lower estimate than the kernel estimate. There was however the interesting finding that entropy was added for keyboard interrupts. This was interesting as the keyboard interrupt then more or less directly result in entropy being added through the `add_input_randomness` method. This means that entropy from keyboard interrupt timings are used twice, as an interrupt and as added input. Briefly looking through the recorded data it is also seen that each time when a keyboard interrupt added randomness there was at least one timer randomness added. The timer randomness added was for most observed data recorded on the same jiffy as the interrupt was recorded. For one of the observed keyboard interrupts no timer randomness was added for the same jiffy, with it instead being added the next jiffy. As most keyboard interrupt and input randomness seems to occur on the same jiffy this means that the entropy from this timing is used twice. In itself this is not necessarily a problem but entropy estimates which do not account for this means that keyboard events are attributed more entropy per timing. As keyboard events usually contain unpredictability in which keys are pressed, this might not necessarily result in entropy overestimates. Depending on what the computer is used for key presses might however be easily guessed or observed. This means that in certain circumstances the event codes will not contain any extra entropy. In combination with the events being used twice this might make overestimates more likely.

8.2.3 Impact

Entropy added is probably larger than the estimates made by both the predictors and the kernel in most circumstances. Entropy estimates are only made of the timings in jiffies while added data usually contains timings with higher precision. This timing from `random_get_entropy` is architecture dependent but on most machines it presumably contains more entropy. As such, any overestimates made by the kernel on entropy contained in jiffies is probably still underestimates of entropy in the higher precision timing.

The fact that the Linux estimator could be fooled to overestimate entropy was known previously. As far as the author knows there had however not been any recorded data where the estimator actually made overestimates during normal usage. Recorded overestimates show that the estimator not only fail on data crafted to subvert it, but also on data from normal operation. The security implications of this are however not very large as more entropy is added through the output of `random_get_entropy` in most architectures. If the source of this data is not trusted as unpredictable or if the architecture is incapable of giving entropy from this function, then it could potentially be a security concern. For `/dev/random` this means that there at some times, where the generator underestimates the entropy, does not provide true random data. The actual effect of this should be small unless an attacker somehow manages to compromise the internal state of the generator. If an attacker at one point had access to the state he will then be able to, with better than random success rate, predict the following random data. Furthermore, the attacker would be able to, with less work than $O(2^n)$ for n output bits, be able to keep track of the internal generator state.

For `/dev/urandom` a failure in the entropy estimation could potentially have a larger impact. The impact on the generator during normal usage should be more or less the same as for `/dev/random` as the generators use the same pool for reseeding. A difference is however that if the internal state is known to an attacker when the ChaCha20 algorithm in `/dev/urandom` is reseeded then the state of the non-blocking generator is known for approximately 5 more minutes, until the next reseed. This means that if the initial seeding of the generator is known, then the state of the generator will remain known for the first 5 minutes after booting. The initial seeding will be done when the estimated entropy is higher than 128 bits and if actual entropy is lower, this directly corresponds to more predictability in the first seeding. If for example the maximum observed underestimate factor f in Table 7.1 was to happen during boot-up, this would correspond to an initial seeding with only $f \cdot 128 > 0.1 \cdot 128 = 12.8$ bits of entropy. This amount of entropy could potentially be brute-forced leading to a state compromise for the first 5 minutes after boot-up.

8.3 FreeBSD Generator

The entropy estimate made in the FreeBSD kernel is provided by the entropy source. In practice most of estimated entropies observed were equal to 1, while some input was estimated to contain 2 bits of entropy. The entropy estimates are not used with the standard configuration of using Fortuna to generate random numbers. When using Yarrow or another module to generate random number the estimates may however be important. As can be seen in Figure 7.13 the estimates made by the kernel seems to be a bit optimistic, when only taking added data into account. When actually adding data to the generators, output from the function `get_cyclecount()` is also added, which gives the cycle count of the CPU. As such, more entropy is added through calls to this function which may mean that actual added entropy still is at least as high as the provided estimate. Analysing the implementation of the `get_cyclecount()` function does however reveal that for certain machines it will return a constant zero [40], presumably because the processor does not provide any cycle count.

Although the Fortuna implementation does not care about entropy estimates, a low entropy rate could still be harmful to the generator. The generator can guarantee that it will eventually reseed the AES block cipher with an unpredictable key. If the entropy rate is low, the time it takes until such a reseeding occurs can be long. The generator attempts to ensure that each reseed is somewhat unpredictable by not reseeding unless at least 64 events have been added to the first pool since the last reseed. If the entropy added for each event is less than 1 bit, this could potentially lead to a reseeding which is relatively predictable to an attacker. The impact of this would be that after initially seeded, the generator will be predictable at least until the next reseed. If enough events occur, it will not take long until reseeding with a pool which is unpredictable enough. As such, even if entropy contained in each event is less than 1 bit the generator would still only be predictable during the first few reseeds. As each event probably contains more than 1 bit of entropy from the `get_cyclecount()` function, the generator can probably be considered secure even directly after the first reseed.

8.4 PRNGD

Entropy gathered by PRNGD depends on which configuration file is used. With a hand–crafted configuration ensuring that programs called all contain unpredictability, the generator should perform well. The example configuration files might however not be suited for all needs, as unpredictability of programs may vary. The output of some programs will be more or less constant causing entropy received from executions after the first to be very low. Other programs may provide output corresponding to a state which an attacker could feasibly monitor, such as network connections. Furthermore, some programs provide output corresponding to the same internal state of the computer as other programs, causing the same entropy to be used twice. The generator as a whole should however be somewhat safe against most attackers. Although each program might contain little to no entropy, the high amount of times the programs are called ensures that the state still is relatively unpredictable. The example configuration also provide quite small estimates on entropy available in output from the executed programs. Furthermore, the generator attempts to add entropy regularly from other cheap to execute programs. The output from these programs is not estimated to contain any entropy at all but is added to the pool anyway. Programs used like this give results such as current time of day, the process id of executing process as well as information about resource usage and execution time.

Entropy estimates from PRNGD are very simple, but as estimated entropy is so low it probably still is an underestimate during most circumstances. The generator does however not really utilize the entropy estimates and could potentially work just as well without them. During initialization the estimates are used to determine when the generator have reached an unpredictable state and thus when it first can be used. After initialization the estimate is not really necessary and provides little to no addition to the functionality of the generator. The only impact the estimate have is on the rate of entropy harvesting, with a low estimated total contained entropy causing more attempts to gather entropy.

The actual generator in PRNGD is somewhat similar to how `/dev/urandom` functioned before switching to using `ChaCha20`. It contains an entropy pool from which output is generated with the SHA–1 hash–function and the pool is updated with the output.

Before and after serving a request for random data the pool is also mixed to ensure that bytes retrieved are influenced by all bits in the pool. The generated data should therefore not leak much information about the pool to an attacker unable to invert SHA–1. If the mixing procedures work as intended with output being mixed back into the pool this should mean that the generator works as a secure PRNG when no entropy is added. Although addition of entropy should not hurt the security of the generator, it should not be necessary if the internal state remains unknown to an attacker. The SHA–1 hash–function do however have vulnerabilities and if able to invert some SHA-1 hashes it might be possible to reconstruct the generators internal state. This would be a big problem if there was not enough entropy added to the generator as then an iterative guessing attack could be performed. Such an attack would cause following outputs to be relatively predictable, and would stay predictable as long as the attacker could observe some outputs. This somewhat validates gathering more entropy when the internal entropy count is low as it increases the entropy rate when an attacker could feasibly recover the state through reversing outputs.

If the state is compromised in any other way the increased collection rate will not directly start. Instead, the process will proceed as normal until the entropy count have decreased below a threshold. This will occur when more data is generated than the estimated entropy being added. Assuming underestimates of entropy, it is precisely in situations like those were iterative guessing attacks are possible. The reason for this being that the generated data then potentially is predictable enough to guess the input entropy. As such increased entropy gathering rate will happen if the attacker is able to perform iterative guessing attacks. The increased entropy rate does however not ensure that an attacker will be unable to perform iterative guessing attacks. An attacker could generate and observe random data, potentially observing more data than entropy gathered, even during the increased gathering rate. This means that added entropy between outputs might still be low enough for an attacker to keep track of the state.

8.5 HAVEGED

The HAVEGED generator seems to leak information about its internal state with every output. With knowledge of previous content of the `RESULT` array, each output directly corresponds to an entry in the `PWALK` array. Previous content of the `RESULT` array is output from the generator the previous time the array was filled. This information is known to an attacker who is able to monitor output of the generator and thus content of the `PWALK` array is leaked with every output. The contents of the array are however modified directly after each output, leaving at least some unpredictability in the array. The modification is however simple consisting only of a rotation and the xor with the input timing. This means that the unpredictability for this position in the array is no larger than the unpredictability of the timing, which the results of Section 7.5.2 showed were no larger than approximately 8 bits.

To update positions in the `PWALK` array, the parameters `pt` and `PT1` are used. These are set to

```
pt  = ((PT >> 18) & 7)
PT1 = ((PT2 >> 28) & 7)
```

which can be found in the code in Figures B.1 and B.3. However, PT2 is less than or equal to ANDPT when PT1 is calculated and ANDPT $< 2^{28}$ meaning that PT1 $= 0$. The positions PT and PT2 in the PWALK array are then updated via the following.

```
PT2 = (((RESULT[(i-8) ^ PT1] ^ PWALK[PT2 ^ PT1 ^ 7]) & ANDPT)
                    & (0xfffffff7)) ^ ((PT ^ 8) & 0x8)
PT = (((RESULT[(i - 8) ^ pt] ^ PWALK[PT ^ pt ^ 7])) &
        (0xffffffef)) ^ ((PT2 ^ 0x10) & 0x10);
```

Most quantities in this update of the position are known or contain only little unpredictability to an attacker who can monitor output results. Contents of the result array is monitored by looking at output random data. Result array position is more or less known with only 8 different possibilities depending on the value of pt. The position in the PWALK array being used is not as predictable, as the previous value of PT and PT2 cannot be considered known to an attacker. It is however easily seen that the position in the array will be the same as the position of one of the most recent outputs. This is seen by noting that both pt and PT1 are smaller than or equal to 7 and thus the xor of them with 7 will still be smaller than or equal to 7. Furthermore, the xor of PT with all numbers smaller than or equal to 7 have already been used in one of the most recent outputs. The same also holds true for PT2. The difference of the current contents of the array and the one since it was output is only the single transformation which added the most recent timing. This means that for each of the 8 choices for pt the value so far contains only one timings worth of entropy. The final modification to the new positions in the walk is done to ensure that the two walks do not point to the same array entries. This is done by setting some bits in the new positions to equal the negation of the other positions value in these bits.

These properties of the generator leads to several vulnerabilities in HAVEGED, with varying requirements, which are presented next.

8.5.1 Knowledge of Timings

An attacker able to monitor the times used as entropy by the program for some time will be able to recover the internal generator state. As position modifications are deterministic when timings are known, the contents of the PWALK array corresponding to output positions are directly available, Contents of the PWALK array are thus recovered with every output from the generator. The attacker also have to recover the positions PT and PT2. As pt is always equal to 0, the updated PT only contain two bits from PT2 unknown to the attacker. Furthermore, PT1 can be any one of 8 different values, and when PT2 is updated one unknown bit from PT is used. This gives that there in total is at most 16 different choices for PT2. These positions can easily be tracked and the number of potential positions decreases when output is generated, until there only is one possible set of positions left. The attacker can then get full knowledge of the generator state by observing output data until all the PWALK array contents are known.

8.5.2 Knowledge of State

An attacker that knows the generator state generator and can monitor generator outputs will be able to partially keep track of the state.

Array contents are altered after every output but in a relatively predictable manner. The 8 numbers used as output from each position are altered with two different timings with other modifications being deterministic. Assuming each timing contains 10 bits of entropy, this means that the contents could be one of approximately 2^{20} different. The timings are not used symmetrically with them being used to modify 3 and 5 numbers respectively. When the position changes, the new position may be recovered by observing the following generated numbers. The 5 numbers at this position that previously were changed with the same timing can be identified. This can be done by, for all 2^{11} positions of 8 number blocks in the array, keep track of the approximately 2^{10} timings which potentially could have altered them. This corresponds to a set of 2^{10} different combinations which these numbers could be now. Assuming that the output actually is uniformly random, each of the 8 numbers generated from this position can be any one of 2^{32} different numbers. Five of these numbers will be part of at least one of the stored sets of 2^{10} array numbers. This set is the one which used to generate the numbers, which identifies the 8 block position used. The order the numbers are generated will then identifies the exact positions. However, this is only possible if only one position could generate these numbers. The probability that all 5 uniformly distributed random numbers can be generated from one set is however low. A total of $2^{32 \cdot 5} = 2^{160}$ number combinations exist while there only is approximately 2^{11} sets and each can only generate approximately 2^{10} different numbers ordered in 8 different ways. There is thus only a negligible chance for the output numbers to be one of the potential outputs from any of the incorrect positions. Upon recovering the new position, the timing used to modify the entry in the `PWALK` array that was used to update the position can also be recovered. The whole `PWALK` entry will not be recovered as the position is smaller than `ANDPT` and one or two bits of the position is decided by the other position. Most of the low order bits will be recovered and as these are the fast changing and unpredictable bits of the timing, most of the timing will be recovered. If the `PT` and `PT2` position updates use array entries modified by both the `HTICK1` and `HTICK2`, this would cause all content at this position to be almost completely known.

The effect of the state being compromised like this is that output will be far from completely random to an attacker. It means that entropy for output random data is about 10 bits, from each selected position. If the position is able to be recovered directly after the first output from this position this means that the next 7 outputs from this position are more or less completely known with at most 10 bits of entropy remaining. As the position in the walks are not updated at the same time it is not possible to observe a value from both positions and then know the positions of the next 14 outputs. Instead, the maximum number of values from known position in a row will be 10. This is achieved after the `PT` position is updated and outputs from the `PT2` position already are known. In case the `PT` position is recovered after the first output from it, there will follow 7 more outputs from known positions before the `PT2` position is updated. The next 3 outputs will however also be from known positions. The first and third of these outputs is from the `PT` position and therefore known. Finally, the second of these outputs is from the non–updated version of `PT2` and will therefore be known. Ten numbers, each 32 bit long are thus output from positions potentially known to an attacker, who could almost completely know the output data. This corresponds to 320 bits of data meaning that 256 bit keys generated with this could potentially be almost completely known to an attacker.

8.5.3 No Previous Knowledge

Without knowledge of the state or the timings of the program it may be possible for an attacker to recover the generator state. Initially nothing is known about the `PWALK` array or the positions `PT` and `PT2`. Observing generator outputs will however, in the same way as in Section 8.5.2 allow content of the `PWALK` array to be partly recovered. The difference is that the actual positions and the array contents initially are unknown. By observing output data the array contents at unknown positions become known. It is however still possible to observe when the same position is used again, in the same way as when the state is known. When the same position is used twice, actual positions can be partly recovered. This is done by identifying which new positions the walk must have gone to when leaving this state previously. This is only few possible choices depending on previous values of `PT` and `PT2` which eventually should allow the positions to be fully recovered. Eventually this will lead to the state being compromised to the same extent as if it was at one point completely known to an attacker. Effects are therefore the same as described in Section 8.5.2.

8.5.4 Impact

The consequence of the vulnerabilities of the generator is that an attacker can predict future output with better probability than chance if able to use the generator. Output is predictable to such an extent that it should be considered insecure if used directly. There does however seem like its entropy gatherer works, and it might be usable as an entropy source. However, as an entropy source it can not be assumed that each output of the generator give full entropy outputs.

One use for HAVEGED is as an entropy source for `/dev/random`, where HAVEGED output is automatically fed into the kernel. This is done when HAVEGED notices that the kernel input entropy pool is no longer full. During entropy estimations, injected data is estimated to be uniformly distributed, causing relatively little data to be necessary to fill the input pool. The function of the Linux generator should however not be compromised, as even without actual unpredictable input the blocking pool should behave similar to the previous, pool based, implementation of `/dev/urandom`. This implementation had no known weaknesses and the addition of somewhat predictable entropy from HAVEGED should thus not be a big security problem.

There is probably no direct attacks against `/dev/random` using HAVEGED as an entropy source. There is however no real reason to use it as `/dev/urandom` will generate numbers without blocking even without it. Furthermore, using HAVEGED as an entropy source removes the one potential benefit of `/dev/random`, the fact that it attempts to function as a TRNG. As HAVEGED generates more data than it has gathered entropy it ensures that the same will be true for `/dev/random`. As it can easily be seen that it makes large overestimates of added entropy, it is no longer possible to claim that `/dev/random` behaves as a TRNG.

8.6 Predictor Modifications

The predictor modification allow entropy analysis of arbitrary data. They overestimate entropy in expectation and gives both a total average estimate for all data and a more details with estimates for every sample. This allows analysis of both the global and local behaviour of the entropy contents. Results are only able to provide an upper bound on the entropy in the data. However, as seen in the results of Section 7.1, resulting overestimates are, for some different types of distributions, relatively close to the actual entropy. The collection predictor also enables better estimates as seen in the results. The Tables 7.1 and 7.2 show that the collection of predictors allow them to work better than any one of its components. Furthermore, the Figures 7.5, 7.6 and 7.7 show how the predictors allow better testing of unknown distributions. Instead of testing all possible predictors one by one until the best match is found the collection allows multiple to be tested at once. The combination still performs better than the predictor which matches the distribution of the generator.

The modifications also allowed the estimators to perform better than the original predictors for uniform data as seen in Figure 7.8. For low entropies they perform similarly, with the modified versions making overestimates and original one making underestimates, as expected. For higher entropies the non-modified versions starts behaving strangely, potentially due to always taking the lowest available estimate. This behaviour is not present in the modified version which continue to make estimates close to the actual entropy for all tested entropies. No comparison was done on more advanced distributions as the original predictors measure min–entropy and the modified versions estimate Shannon–entropy, meaning they only provide same estimate on uniform data.

8.7 Final Conclusions

Multiple different random number generators with entropy input exist and can be used during different circumstances. Some are built in directly into the operating system while some run as user–space programs. Analysed generators all had some entropy related behaviour that seemed a bit strange. The FreeBSD generator had almost completely removed entropy estimates from its functionality, but estimates were still available. These estimates were shown to be overestimates of the entropy in the data being added, which could have been a security vulnerability. However, the generator added more data afterwards, whose entropy, although not analysed, should be larger than estimated entropy. Furthermore, as entropy estimates were not used during normal usage, they being overestimates would have no real impact on the security of the generator. The generator in the Linux kernel was also shown to occasionally overestimate entropy added from the timing of data in jiffies. As the data most of the time is accompanied by a higher precision timing, this will usually not actually overestimate added entropy. In the presumably rare occasions where no high precision timer is available and entropy added is overestimated, the security risk should still not be large. The only problems are related to if the internal state was compromised recently or if the computer recently started. Furthermore, the `/dev/urandom` implementation ignores the entropy estimate after being initialized and will thus only be impacted by entropy overestimates when initializing.

The software generators analysed seem to have vulnerabilities that potentially could severely impact the security of the generators. The `PRNGD` generator seems to potentially overestimate entropy as it may add identical data multiple times as new entropy. Some data it uses as entropy might be monitored by attackers, causing the data to be predictable for an attacker. By configuring the generator with sources specific to the computer running the generator, it should be possible to make it secure. The HAVEGED generator does however seem to have some real problems when used as a generator. It does seem to generate entropy, but fails to generate unpredictable numbers. By trying to expand the gathered entropy into more randomness in a non–secure manner, the generator instead generates relatively predictable numbers. This makes the generator unfit to be used by itself, but could in theory still be used as entropy source. As a well seeded PRNG or generators available in operating systems in most circumstances perform better, there is however little reason to use HAVEGED as an entropy source.

Vulnerabilities presented in this thesis, mainly the ones related to HAVEGED, could potentially be exploited by attackers. When vulnerabilities are found in important well–used products, it may be questionable to present them to the public. Instead, it might be better to present them to the product developer, allowing them to fix the vulnerability before it becomes well–known. The HAVEGED vulnerabilities are however not very impactful for most people using the generator as an entropy source for other generators. It is also possible for users to stop using the generator if concerned about its security. Presenting the found vulnerabilities thus allow users to get better information about the generator's weaknesses, allowing them to use that information to decide if to use it. For other generators, the vulnerabilities found are so small that they should have little impact on most users, as discussed in the thesis. Presenting these result will probably not lead to vulnerabilities being exploited, but could potentially cause the generators to be updated to avoid similar vulnerabilities.

8.8 Future Work

There was a regular pattern found when investigating the overestimates made by the Linux kernel. This pattern was present in all the maximal overestimates the predictors made but had multiple different codes indicating their source. As such, there seems to be multiple different processes creating a predictable timing pattern. No further investigation was done into why this was the case and what these processes were. Knowledge about this would be interesting but would require more work to pinpoint the sources.

Furthermore, it might be of interest to analyse how the computer architecture impacts the entropy available in the kernel. With an architecture where the Linux kernel is unable to add more entropy from the cycle count, any overestimates could be actual security vulnerabilities. Therefore, tests with such a machine would be interesting to potentially find actual vulnerabilities.

More analysis of the FreeBSD generators functionality may also be of interest. In this thesis it was shown that the entropy estimates that the kernel made were overestimates. Although, as discussed, this has no real impact, this still raises the question about how quickly the generator is seeded with an unpredictable seed. Analysis of entropy contents of the different Fortuna pools when reseeding and the frequency of reseeds might thus be of interest.

Finally, a comparison in performance between the developed entropy estimators and previously available estimators would be interesting. The developed entropy estimator allowed a type of analysis that other tools were unable to provide. Their global average estimates for sources could however be compared with the entropy estimates provided by other entropy estimators. Furthermore, more testing on sources with known distributions could be useful to get a better understanding of how well the method actually functions.

Bibliography

[1] O. Goldreich. *Foundations of Cryptography: Basic Techniques*. Foundations of Cryptography: Basic Tools. Cambridge University Press, 2001. ISBN 9780521791724. URL `https://books.google.se/books?id=ladKNAEACAAJ`.

[2] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001. ISSN 1559-1662. doi: 10.1145/584091.584093. URL `http://doi.acm.org.focus.lib.kth.se/10.1145/584091.584093`.

[3] John Walker. A pseudorandom number sequence test program. *Open Source software library, under development*, 2008.

[4] Cédric Lauradoux, Julien Ponge, and Andrea Röck. *Online entropy estimation for non-binary sources and applications on iPhone*. PhD thesis, INRIA, 2011.

[5] Patrick Lacharme, Andrea Rock, Vincent Strubel, and Marion Videau. The Linux Pseudorandom Number Generator Revisited, 2012. URL `https://hal.archives-ouvertes.fr/hal-01005441`. déposé sur Cryptology ePrint Archive (http://eprint.iacr.org/).

[6] Sarit Shwartz, Michael Zibulevsky, and Yoav Y Schechner. Fast kernel entropy estimation and optimization. *Signal Processing*, 85(5):1045–1058, 2005.

[7] Elaine B Barker and John Michael Kelsey. *(Second DRAFT) Recommendation for the Entropy Sources Used for Random Bit Generator*. US Department of Commerce, National Institute of Standards and Technology, 2016.

[8] John Kelsey, Kerry A. McKay, and Meltem Sönmez Turan. Predictive models for min-entropy estimation. *IACR Cryptology ePrint Archive*, 2015:600, 2015.

[9] Robert G Brown, Dirk Eddelbuettel, and David Bauer. Dieharder: A random number test suite. *Open Source software library, under development*, 2017.

[10] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, August 2007. ISSN 0098-3500. doi: 10.1145/1268776.1268777. URL `http://doi.acm.org/10.1145/1268776.1268777`.

[11] Aaron D Wyner and Jacob Ziv. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. *IEEE Transactions on Information Theory*, 35(6):1250–1258, 1989.

[12] Michael Kerrisk. *RANDOM(4) Linux Programmer's Manual*, January 2017.

[13] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[14] François Goichon, Cédric Lauradoux, Guillaume Salagnac, and Thibaut Vuillemin. *Entropy transfers in the Linux random number generator*. PhD thesis, INRIA, 2012.

[15] Khudran Alzhrani and Amer Aljaedi. Windows and linux random number generation process: A comparative analysis. *International Journal of Computer Applications*, 113(8), 2015.

[16] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the random number generator of the windows operating system. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):10, 2009.

[17] Expected value, 2017. URL `https://en.wikipedia.org/wiki/Expected_value`.

[18] Conditional probability, 2017. URL `https://en.wikipedia.org/wiki/Conditional_probability`.

[19] Independent and identically distributed random variables, 2017. URL `https://en.wikipedia.org/wiki/Variance`.

[20] Markov chain, 2017. URL `https://en.wikipedia.org/wiki/Markov_chain`.

[21] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[22] Information theory, 2017. URL `https://en.wikipedia.org/wiki/Information_theory`.

[23] Ergodic process, 2017. URL `https://en.wikipedia.org/wiki/Ergodic_process`.

[24] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.

[25] Cryptography Research Inc. Evaluation of via c3 nehemiah random number generator, 2003. URL `http://www.cryptography.com/public/pdf/VIA_rng.pdf`.

[26] André Seznec and Nicolas Sendrier. Havege: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, October 2003. ISSN 1049-3301. doi: 10.1145/945511.945516. URL `http://doi.acm.org.focus.lib.kth.se/10.1145/945511.945516`.

[27] Lutz Jänicke. Prngd - pseudo random number generator daemon, 2007. URL `http://prngd.sourceforge.net/`.

[28] Mads Haahr. Random. org: True random number service. *School of Computer Science and Statistics, Trinity College, Dublin, Ireland. Website (http://www. random. org). Accessed*, 1, 2017.

[29] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *International Workshop on Fast Software Encryption*, pages 168–188. Springer, 1998.

[30] Theodore Ts'o. Linux kernel commit. URL `https://github.com/torvalds/linux/commit/e192be9d9a30555aae2ca1dc3aad37cba484cd4a`.

[31] Linux kernel, feb 2017. URL `https://github.com/torvalds/linux`.

[32] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *International Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999.

[33] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003. ISBN 0471223573.

[34] haveged - a simple entropy daemon, 2017. URL `http://www.issihosts.com/haveged/`.

[35] André Seznec and Nicolas Sendrier. HArdware Volatile Entropy Gathering and Expansion: generating unpredictable random number at user level. Research Report RR-4592, INRIA, 2002. URL `https://hal.inria.fr/inria-00071993`.

[36] Abraham J Wyner and Dean Foster. On the lower limits of entropy estimation. 2003.

[37] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[38] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[39] Yun Gao, Ioannis Kontoyiannis, and Elie Bienenstock. Estimating the entropy of binary time series: Methodology, some theory and a simulation study. *Entropy*, 10(2): 71–99, 2008.

[40] Freebsd source, mar 2017. URL `https://github.com/freebsd/freebsd`.

[41] CISSP CSSLP W. Dean Freeman. Deeper exploration in (free|hardened)bsd entropy (part 2 of series). URL `https://www.weaponizedawesome.com/blog/?p=173`.

[42] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2015. Last accessed 2017-03-20.

[43] GNU. libconfig - c/c++ configuration file library, 2017. URL `http://www.hyperrealm.com/libconfig/`.

[44] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. Technical report, 2015.

[45] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, 2008.

Appendix A

ChaCha20 Generator in Linux

ChaCha20 is a stream cipher which when given a 256 bit key, a nonce and a counter produce a random looking stream that can be combined with data to encrypt it. As the stream is meant to be random when working as a stream cipher, it can also be used as a random generator. The internal state of the ChaCha20 cipher consists of an array 16 numbers, with each number 32 bits long. The first 4 of these numbers are constants that are the same for all implementations of ChaCha20. Remaining are the 8 numbers of the key and 4 numbers consisting of the nonce and the block counter. The original implementation keeps only one 32 bit number for block counter while the remaining 96 bits are used for the nonce [44].

The implementation of the random number generator in Linux instead uses a 64 bit counter that is incremented every time random data is generated. Every time data is modified, the 64 bit nonce part of the state is also potentially modified, although how and if it is altered depends on the computer architecture. This is done by updating the nonce to be the exclusive or of the previous nonce and a number from a function called `arch_get_random_long`. With x86 processors this attempts to use the RdRand instruction to get random data while on other architectures it seems to not generate output.

When the generator is initialized it sets the first 4 numbers to the constant values they are supposed to be and attempts to set the remaining 12 numbers in the state to be random. To do this, the numbers are first set to equal entropy extracted from the input pool. Afterwards, the xor of the state and randomness from sources that depend on computer architecture is set to be the new state. The initialization is done early during the booting process and the state of the input pool is therefore not considered unpredictable at this point. Therefore, the generator is not considered seeded and usable until more entropy have been added. This is done by reseeding as soon as the estimated amount of entropy in the input pool is above 128 bits and by adding entropy from interrupts to the key portion of the generator. When reseeding, the generator extract entropy from an entropy pool and updates the key part of the generator state [31].

The actual algorithm to generate output is based upon 20 rounds of operations on the data. Each round consists of 4 quarter rounds where each quarter round operates on 4 of the 16 numbers in the state. Denoting the four numbers in a quarter round a, b, c, d the

Round	a	b	c	d
1	0	4	8	12
2	1	5	9	13
3	2	6	10	14
4	3	7	11	15
5	0	5	10	15
6	1	6	11	12
7	2	7	8	13
8	3	4	9	14

Table A.1: The index of the chosen numbers in the array for each quarter round

operation can be expressed as

$$a = a + b;\ \ d = d \oplus a;\ \ d <<<= 16; \tag{A.1}$$

$$c = c + d;\ \ b = b \oplus c;\ \ b <<<= 12; \tag{A.2}$$

$$a = a + b;\ \ d = d \oplus a;\ \ d <<<= 8; \tag{A.3}$$

$$c = c + d;\ \ b = b \oplus c;\ \ b <<<= 7; \tag{A.4}$$

Here $\oplus$ is exclusive or and $x <<<= n$ is rotation of the binary number by n bits towards the high order bits. The choices of a, b, c, d for each quarter round is shown in Table A.1 for the first 8 quarter rounds. This configuration is then repeated for every following pair of rounds giving a total of 10 pairs of rounds each following this configuration [44, 45].

Appendix B

HAVEGED code

The code used to generate random numbers in HAVEGED is shown here. The code is modified slightly with parts whose only purpose is to alter the running time removed and some statements moved in ways which do not affect behaviour. For description of code function, see Section 7.5.1.

```
HARDCLOCKR(HTICK1);

pt = (PT >> 18) & 7;

PTTEST = PT >> 20;

PT = PT & ANDPT;

Pt0 = &PWALK[PT];
Pt1 = &PWALK[PT2];
Pt2 = &PWALK[PT  ^ 1];
Pt3 = &PWALK[PT2 ^ 4];

RESULT[i++] ^= *Pt0;
RESULT[i++] ^= *Pt1;
RESULT[i++] ^= *Pt2;
RESULT[i++] ^= *Pt3;

inter = ROR32(*Pt0,1) ^ HTICK1;
*Pt0  = ROR32(*Pt1,2) ^ HTICK1;
*Pt1  = inter;
*Pt2  = ROR32(*Pt2, 3) ^ HTICK1;
*Pt3  = ROR32(*Pt3, 4) ^ HTICK1;

Pt0 = &PWALK[PT  ^ 2];
Pt1 = &PWALK[PT2 ^ 2];
Pt2 = &PWALK[PT  ^ 3];
Pt3 = &PWALK[PT2 ^ 6];

RESULT[i++] ^= *Pt0;
RESULT[i++] ^= *Pt1;
RESULT[i++] ^= *Pt2;
RESULT[i++] ^= *Pt3;
```

Figure B.1: The generation of the first 8 numbers from the `oneiteration.h` file

```
if (PTTEST & 1) {
  PTTEST ^= 3; PTTEST = PTTEST >> 1;
  if (PTTEST & 1) {
    PTTEST ^= 3; PTTEST = PTTEST >> 1;
    if (PTTEST & 1) {
      PTTEST ^= 3; PTTEST = PTTEST >> 1;
      if (PTTEST & 1) {
        PTTEST ^= 3; PTTEST = PTTEST >> 1;
        if (PTTEST & 1) {
          PTTEST ^= 3; PTTEST = PTTEST >> 1;
          if (PTTEST & 1) {
            PTTEST ^= 3; PTTEST = PTTEST >> 1;
            if (PTTEST & 1) {
              PTTEST ^= 3; PTTEST = PTTEST >> 1;
              if (PTTEST & 1) {
                PTTEST ^= 3; PTTEST = PTTEST >> 1;
                if (PTTEST & 1) {
                  PTTEST ^= 3; PTTEST = PTTEST >> 1;
                  if (PTTEST & 1) {
                    PTTEST ^= 3; PTTEST = PTTEST >> 1;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
};
PTTEST = PTTEST >> 1;
if (PTTEST & 1) {
  Ptinter = Pt0;
  Pt2 = Pt0;
  Pt0 = Ptinter;
}
```

Figure B.2: First use of `PTTest` variable

```
inter  = ROR32(*Pt0, 5) ^ HTICK1;
*Pt0   = ROR32(*Pt1, 6) ^ HTICK1;
*Pt1   = inter;
HARDCLOCKR(HTICK2);
*Pt2 = ROR32(*Pt2, 7) ^ HTICK2;
*Pt3 = ROR32(*Pt3, 8) ^ HTICK2;

Pt0 = &PWALK[PT  ^ 4];
Pt1 = &PWALK[PT2 ^ 1];
PT2 = (RESULT[(i - 8) ^ PT1] ^ PWALK[PT2 ^ PT1 ^ 7]);
PT2 = ((PT2 & ANDPT) & (0xfffffff7)) ^ ((PT ^ 8) & 0x8);
PT1 = ((PT2 >> 28) & 7);
Pt2 = &PWALK[PT  ^ 5];
Pt3 = &PWALK[PT2 ^ 5];

RESULT[i++] ^= *Pt0;
RESULT[i++] ^= *Pt1;
RESULT[i++] ^= *Pt2;
RESULT[i++] ^= *Pt3;

inter = ROR32(*Pt0 , 9) ^ HTICK2;
*Pt0  = ROR32(*Pt1 , 10) ^ HTICK2;
*Pt1  = inter;
*Pt2  = ROR32(*Pt2, 11) ^ HTICK2;
*Pt3  = ROR32(*Pt3, 12) ^ HTICK2;

Pt0 = &PWALK[PT  ^ 6];
Pt1 = &PWALK[PT2 ^ 3];
Pt2 = &PWALK[PT  ^ 7];
Pt3 = &PWALK[PT2 ^ 7];

RESULT[i++] ^= *Pt0;
RESULT[i++] ^= *Pt1;
RESULT[i++] ^= *Pt2;
RESULT[i++] ^= *Pt3;

inter = ROR32(*Pt0, 13) ^ HTICK2;
*Pt0  = ROR32(*Pt1, 14) ^ HTICK2;
*Pt1  = inter;
*Pt2  = ROR32(*Pt2, 15) ^ HTICK2;
*Pt3  = ROR32(*Pt3, 16) ^ HTICK2;

PT = (((RESULT[(i - 8) ^ pt] ^ PWALK[PT ^ pt ^ 7])) &
      (0xffffffef)) ^ ((PT2 ^ 0x10) & 0x10);
```

Figure B.3: Generation of the remaining 8 values from one iteration