

BACHELOR THESIS  
COMPUTER SCIENCE



RADBOD UNIVERSITY

---

# The State of Entropy Generation in Practice

---

*Author:*  
Hendrik Werner  
s4549775

*First supervisor/assessor:*  
dr. Veelasha Moonsamy  
email@veelasha.org

*Second supervisor:*  
prof. dr. Erik Poll  
erikpoll@cs.ru.nl

*Second assessor:*  
dr. Asli Bay  
a.bay@cs.ru.nl

April 1, 2019

## Abstract

Cryptographically secure random number generation using deterministic computers is a difficult problem. Pseudo-random number generation algorithms with mathematically proven properties exist, but small implementation errors can lead to disastrous security vulnerabilities. Additionally, generating high entropy randomness for seeding these algorithms remains an open problem. While best practices for entropy generation exist, they are often badly implemented in practice, or completely ignored. Discourse about these issues is hindered by the fact that the terms *entropy* and *randomness* are often used inconsistently, and without properly defining them, which we also try to rectify by meticulously stipulating the definitions used in the thesis.

This thesis summarizes the known best practices, and assesses entropy generation failures in the wild, based on representative examples taken from the open source ecosystem. It is demonstrated that entropy generation remains problematic in practice, despite being an important and theoretically well studied discipline; in part due to the ignorance of implementors who disregard best practices, and in part because it remains an inherently difficult problem. Advice on how to securely generate random numbers is presented. However, just knowing about best practices is not sufficient, and to effectively improve the state of entropy generation in practice, automated code reviews are required.

A template for this automation process is provided in the form of QL queries, which can be used to eliminate vulnerabilities by means of *variant analysis* and *data flow analysis*. The effectiveness of this approach is demonstrated by analyzing about 8000 C/C++ projects on the LGTM code analysis platform, which yields a large number of real vulnerabilities, allowing the state of entropy generation in practice to be determined, and hopefully improved.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The Problem . . . . .	3
1.2	Contributions . . . . .	4
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
2.1	Entropy . . . . .	6
2.2	Noise Source . . . . .	8
2.3	Entropy Source . . . . .	9
<b>3</b>	<b>Literature Review</b>	<b>11</b>
3.1	Entropy . . . . .	11
3.2	Entropy Generation . . . . .	12
3.3	Entropy Attacks . . . . .	13
<b>4</b>	<b>Examples</b>	<b>15</b>
4.1	Hacker News Login Cookie Vulnerability . . . . .	15
4.2	Widespread Weak Cryptographic Keys . . . . .	15
4.3	Badly Implemented Random Number Generator in 7-Zip . . . . .	16
4.4	Unsuitable Noise Source in the Godot Game Engine . . . . .	19
<b>5</b>	<b>Entropy Generation Best Practices</b>	<b>22</b>
5.1	High Entropy Noise Sources . . . . .	22
5.2	Multiple Noise Sources . . . . .	24
5.3	Conditioning . . . . .	24
5.4	Health Testing . . . . .	25
5.5	Advice for Implementors . . . . .	26
<b>6</b>	<b>Entropy Generation in Practice</b>	<b>28</b>
6.1	Predictable Noise Sources . . . . .	29
6.2	Single Noise Source . . . . .	29
6.3	No Conditioning . . . . .	29
6.4	Missing Health Tests . . . . .	30
<b>7</b>	<b>Detecting Entropy Generation Weaknesses</b>	<b>32</b>
7.1	Database extraction . . . . .	33
7.1.1	Importing Godot . . . . .	35
7.2	Detecting Unsuitable Noise Sources . . . . .	35
7.3	Data Flow Analysis . . . . .	41
7.4	Analysis Results . . . . .	44
<b>8</b>	<b>Conclusions</b>	<b>51</b>
<b>9</b>	<b>Appendix</b>	<b>52</b>
9.1	A: Proof of Concept . . . . .	52
9.2	B: Emails from Christian Uldall Pedersen . . . . .	53
9.3	C: LGTM Sample Projects . . . . .	55

# 1 Introduction

Random numbers are required for a number of applications, such as statistical analysis (e.g. the bootstrap method), statistical sampling, physics and computer science (e.g. Monte Carlo methods), and cryptography [7, 20, 18, 24, 32].

Randomness is hard to describe, and relative to the domain it is used in. Some interpretations even say that no general notion of randomness can exist. We cannot go into much detail here, as that would fall out of the scope of this thesis. We can say however, that what we essentially expect of a random sequence is that we cannot reliably predict it [12, 19, 22, 64, 24].

Algorithms on the other hand, are inherently deterministic. Given the same input, they always produce the same output. That means that by definition it is impossible to create “true” randomness using an algorithm, and by extension a computer [42]. This leaves us in a difficult situation. Randomness is hard to define and produce, but required for many tasks.

Thankfully, most applications of randomness do not require “true” random sequences, but rather they require these sequences to have certain properties [24]. In some areas it is even undesirable to use “true” randomness, e.g. due to a lack of repeatability [49]. Therefore, the problem of random number generation has largely been solved by pseudo-random number generators (PRNGs). There are countless known PRNG algorithms (e.g. Mersenne Twister [39] or Permuted Congruential Generator (PCG) [43]), and all of them share the same property: given an initial state called *seed*, they will produce a sequence that adheres to some set of properties. These properties (e.g. period length) can be mathematically proven [41, 39, 43].

The PRNG algorithms are still deterministic however, which means that given the seed, the sequence of pseudo-random numbers can be reproduced. This is because as discussed above, these numbers are not actually random, they are determined by a PRNG algorithm based on the seed.

Polymath John Von Neumann had the following to say on the matter:

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” [42]

PRNGs solve the problems of repeatability and cheap generation of large sequences of numbers which appear random according to some set of criteria. One disadvantage of these algorithms is that they leave open the question of how to initialize (a.k.a. seed) them.

## 1.1 The Problem

In theory it is well known that the seeds used in PRNGs must be chosen very carefully, and should be kept confidential. If an attacker knows the seed, the PRNG output can be predicted, and the security of everything that uses the randomness generated from this seed is compromised [7, 43].

To prevent attackers from guessing the seed, it has to contain high entropy (section 2.1). As discussed earlier, generating entropy is not an easy task for computers. There are known best practices for collecting entropy on deterministic computers [70, 7, 68]. These techniques are, for example, implemented in the Linux kernel’s `/dev/random` device. `/dev/random` uses low level hardware information like clock jitter and user input to collect entropy in an entropy pool, and contains an estimator for the amount of entropy it has gathered [41]. Estimating entropy is an exceptionally hard task, and previous research [18, section 5.4] has shown that in certain situations

the entropy estimator of `/dev/random` can be fooled. This means that even following the known best practices does not necessarily prevent exploits based on insufficient entropy [70].

In practice however, the situation is even worse, and not just limited to theoretical attack scenarios. An attack vector being well known and often exploited does not mean that it will no longer occur in the real world [21, 32, 31]. As an example of this, let us consider buffer overflows. We have known about them at least since 1962 [8], and there are many known techniques to prevent them. These techniques include bounds checking and formal verification. Despite the attack and prevention techniques being old and well known, it still occurs in the wild to this day [16, 15].

## 1.2 Contributions

The main goal of this thesis is to determine, and hopefully improve, the state of entropy generation in practice. This task is broken down into its constituents, which are tackled individually.

In section 2, the first hurdle is taken on, which consists of meticulously outlining the definitions of entropy and related concepts like *entropy sources*, and *noise sources*, which are used in this thesis. It is essential to be on the same level about these concepts, and to use consistent terminology, if the goal is to have a constructive conversation.

The existing rich literature about entropy, how to generate entropy, vulnerabilities arising from entropy generation, and relevant background information is discussed in section 3. We review a representative cross section of (academic) research, and how it related to this thesis.

Section 4 covers two recent, and two historical (though still applicable) examples of entropy generation failures in practice, which are being referred to throughout the remainder of the thesis. Explanations of why each example was picked are included in the section itself. A common deficit of the existing literature is the purely theoretical nature of their findings, which are not applied to practical scenarios. A lot of the time, researchers are content with proving some construction theoretically unsound according to their security model, without so much as a proof of concept. We try to address this shortcoming by using real world examples.

All the information about entropy generation best practices, which was scattered throughout the literature, is compiled into an easy to follow checklist in section 5, which can be used as a guide for somebody wanting to implement a secure entropy source. The NIST SP 800-90B [68] technical report was the closest thing to such a manual that could be found in the existing literature, but it is very technical, and assumes a lot of prior knowledge in order to be properly understood and followed. Not everybody has the capacity and/or inclination to become an expert on entropy generation in order to implement an entropy generator. This section improves the status quo by providing a succinct and easily understood checklist of best practices to follow, describing what should be done, why it should be done, and how it can be implemented.

As a counterpart to section 5, section 6 demonstrates how several of the best practices are not applied in many real world cases during the construction of entropy sources, and how this has led and will lead to countless security vulnerabilities in the field. Common errors and pitfalls encountered by implementors of entropy sources are listed, which implementors should check their implementation against, after building an entropy generator according to the instructions in section 5. We refer back to the examples from section 4, which made a lot of errors when constructing their entropy sources.

Judging from hundreds of years of practical experience, it can be said with confidence that humans are naturally lazy, and will take shortcuts and cut corners whenever they see the possibility to do so. This means that having a checklist of best practices, as well as a list of common errors and pitfalls is a great start, but not sufficient to sustainably improve the state of entropy generation in practice. Even if a codebase was cleaned up 100%, following all the best practices, and removing all flaws (which is a hopeless endeavor in most cases), errors would seep back in over time. Ideally the best practices should be automatically enforced by code review tools.

Based on the observations made in sections 4, 5, and 6, section 7 takes a step towards realizing this goal. QL queries attempting to automatically detect flaws in entropy generation with *variant analysis* (section 7) and *data flow analysis* (section 7.3) are developed, which can be run using the LGTM code analysis platform. Bas van Schaik, who works for the company behind LGTM, kindly offered to run the query against their whole catalogue of about 8000 diverse C/C++ projects. The results of this analysis are discussed, conveying an overview about the state of entropy generation in practice.

## 2 Preliminaries

In many discussions about randomness and entropy, these terms are not defined satisfactorily, and thus they are often used contradictorily. By properly defining terms and functions which are used throughout this thesis, this section is intended to preempt confusion, and to foster a constructive dialogue that is not hindered by ambiguities, communication problems, or misunderstandings.

### 2.1 Entropy

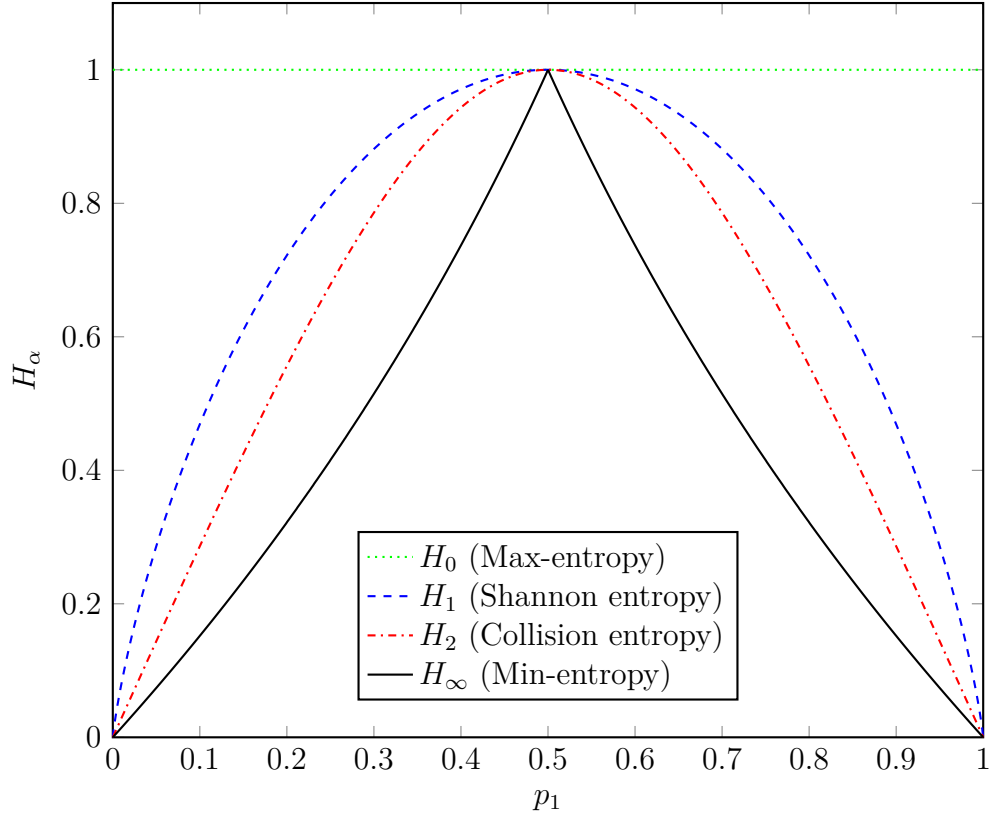


Figure 1: Rényi entropy measures

“No one knows what entropy really is, so in a debate you will always have the advantage.”  
(John von Neumann to Claude Shannon) [67]

Entropy is a versatile and fluid concept with applications in many different areas, ranging from thermodynamics to Information Theory, and cryptography. Naturally, this makes it hard to pinpoint a generic definition, and hinders discourse. In the following, a short overview of the history of entropy is given. The section concludes with the definitions and formulas pertaining to entropy, which are used in this thesis.

Initially, entropy was introduced in 1865 by Rudolf Clausius in the field of classical thermodynamics [13]. Ludwig Boltzmann later discovered the connection with statistical probability, and Max Planck wrote his formula in the way it is used today:  $S = k_B \ln W$ , where  $k_B$  is the Boltzmann constant. J. Willard Gibbs generalized Boltzmann’s entropy into a form that already looks rather familiar:  $S = -k_B \sum_i p_i \ln p_i$ .

Following the deployment of large parts of the American national communications network, several engineering departments of *American Telephone & Telegraph (AT&T)* and the *Western Electric* company were consolidated into *Bell Telephone Laboratories, Inc.* in 1925 [33]. Formerly tasked with overcoming the day-to-day engineering challenges of building such a communications network, the researchers were now dedicated to fundamental research, which resulted in the discovery of Information Theory, and ultimately led to a breakthrough in the field of statistical entropy.

This began with the article “Transmission of Information” by Ralph V. L. Hartley [29] published in 1928, in which he sets out a quantitative measure of the capacity of information that can be transmitted by a system. Even though Hartley’s function  $H = \log |X|$  can be interpreted in retrospect as a measure of entropy for equidistributed stochastic processes with  $|X|$  outcomes, and is indeed sometimes called Hartley entropy, it was derived without introducing probability. Today, Hartley entropy is often referred to as “Max-entropy”, because it assumes a uniform probability distribution, which coincides with the maximum entropy according to every Rényi entropy measure (see below).

In 1948, Claude Shannon finally generalized entropy away from physics, in order to apply it to Information Theory. Shannon entropy measures the average rate at which information is generated by stochastic processes. It is still in use today, and calculated as follows:  $H = -K \sum_i p_i \ln p_i$ , where  $K$  is a positive constant (often  $K = 1$ , as “ $K$  merely amounts to a choice of a unit of measure”) [62]. Alfréd Rényi further generalized Shannon entropy, as well as Hartley entropy (a.k.a. Max-entropy), Collision entropy, and Min-entropy into a new very general entropy measure called Rényi entropy:  $H_\alpha = \frac{1}{1-\alpha} \log(\sum_i p_i^\alpha)$ , where  $\alpha \geq 0, \alpha \neq 1$  (figure 1) [57].

Although statistical entropy is important in many different areas, it remains elusive, as indicated by the quote from John von Neumann [67]. The main reason this thesis is concerned with entropy is that it is needed to provide (security) guarantees for systems based on randomness, such as many cryptosystems. First of all, a distinction has to be made between entropy of a random process, and entropy of a sequence.

- Entropy for random processes is easier to define. The statistical entropy of a random process is the amount of indeterminacy of its outcomes. Each toss of a fair coin is more random, and therefore has higher entropy, compared with a biased coin. Peak entropy is reached when no strategy of guessing the outcome gives an attacker an advantage. This is the case for uniform probability distributions, where  $\forall_{i,j}. p_i = p_j$ .
- Defining entropy for random sequences is harder, and there may not even be such a thing as a random sequence [12, 19]. Looking at the example of coin tosses again, is the sequence *HHHHHHHHHH* any less random than the sequence *TTHHTTHTTT*? Both sequences can be produced by flipping a fair coin (the latter sequence was actually produced in this way), and they are even equally likely to occur. The first sequence, consisting solely of heads, can be described more concisely, and is therefore less random according to some entropy measures. However, there is no real consensus if this statement even makes sense. Another viewpoint is that no general notion of random sequences can exist [64].

In addition to this dichotomy, there are several competing entropy measures for each interpretation of entropy. Following the example of NIST SP 800-90B [68], this thesis will focus on min-entropy. Being the lowest of the Rényi entropy measures ( $\alpha = \infty$ ) (figure 1), it should give the highest confidence in the guarantees given by systems based on it.



Figure 1 plots different Rényi entropy measures with different values of  $\alpha$  against the probability  $p_1$ , for a random binary process with outcomes  $x_1$  and  $x_2$ .  $p_1$  denotes the probability of outcome  $x_1$  occurring, and  $p_2 = 1 - p_1$  correspondingly denotes the probability of outcome  $x_2$ . This is intentionally kept generic, and is therefore applicable to many different situations. For example, for a fair coin toss,  $x_1 = H, x_2 = T$ , and  $p_1 = p_2 = \frac{1}{2}$ .

$$\begin{aligned} H_\infty &= \min_i (-\log_2 p_i) \\ &= -\log_2 (\max_i p_i) \end{aligned}$$

**Example:** Each fair coin toss has two outcomes, which both occur with probability  $\frac{1}{2}$ , therefore  $\max_i p_i = \frac{1}{2}$ . Min-entropy tells us that each fair coin toss contains a single bit of entropy:

$$\begin{aligned} H_\infty &= -\log_2 \frac{1}{2} \\ &= 1 \end{aligned}$$

One would expect a biased coin to have less entropy per toss. If a coin lands heads up with probability  $\frac{2}{3}$ , then  $\max_i p_i = \frac{2}{3}$ .

$$\begin{aligned} H_\infty &= -\log_2 \frac{2}{3} \\ &\approx 0.585 \end{aligned}$$

This result confirms the intuition that biased coins have less entropy per toss, compared with fair coins. As expected, a uniform probability distribution leads to the highest uncertainty, and thus the highest entropy.

## 2.2 Noise Source

Noise sources are the core of every entropy source, and are ultimately responsible for providing the unpredictability. If the noise source fails to produce entropy, then this lack of indeterminacy cannot be compensated for by any other component. This makes noise sources the core of pseudo random number generation. PRNGs need an entropy source, and entropy sources need at least one noise source [68].

In NIST SP 800-90B and this thesis, it is assumed that noise sources provide digital output. If a non-digital process is sampled, the noise source must contain a digitization step (figure 2). The digital output is called *raw data*.

In principle, every non-deterministic process with inherent randomness can act as a noise source. There are two categories of noise sources [68]:

1. Physical noise sources, like radioactive decay, background radiation, accelerometer data, etc., which typically provide high quality randomness, but require specialized hardware to capture.

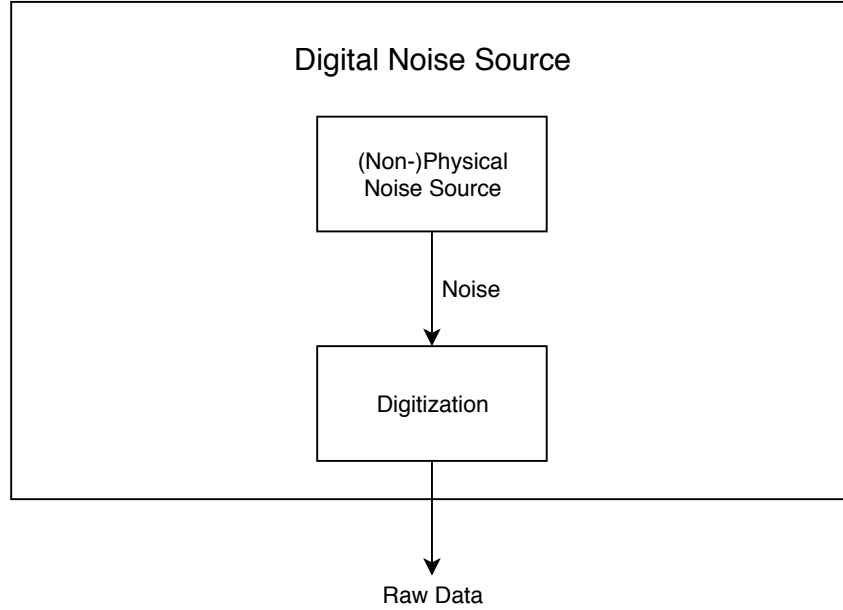


Figure 2: NIST SP 800-90B Noise Source

2. Non-physical noise sources, like clock jitter, network packet arrival times, interrupt timings, and user input (e.g. mouse and keyboard), which rely on system data, and sampling them does not require specialized hardware. They are easier to sample, but may provide lower quality randomness, or inconsistent randomness, which can lead to a lack of entropy, especially in embedded systems, which have less noise sources available [32].

For many processes it is not obvious how much entropy they provide, and how consistent the resulting entropy is. It is important to have a good understanding of the process which is used as a noise source, in order to predict the output entropy, as well as possible failure states. If a noise source fails to provide enough entropy, no security guarantees can be provided by systems relying on it, and it may not be immediately obvious that it failed [68].

## 2.3 Entropy Source

Entropy sources are the main focus of this thesis. They depend on a noise source, and provide random output with a guaranteed minimum entropy. In addition to at least one noise source, NIST SP 800-90B [68] entropy sources contain an optional conditioning component (section 5.3), as well as mandatory health tests (section 5.4) (figure 3).

*Raw data* is taken from the noise source, and fed into the conditioner, as well as into the health tests. Conditioning is optionally used to increase entropy per bit, and/or decrease bias in the *raw data*. The conditioner outputs random bits with a specified minimum amount of entropy. Health tests ensure the continued functionality of the noise source. If a failure is detected, a corresponding error is output, and the entropy source refuses to provide further random bits.

NIST SP 800-90B [68] assumes that entropy source implementors make a good faith effort to implement entropy sources which provide consistent entropy, meeting or exceeding an entropy threshold. As demonstrated by multiple sources, this is not always a valid assumption to make (section 4) (section 6) [32].

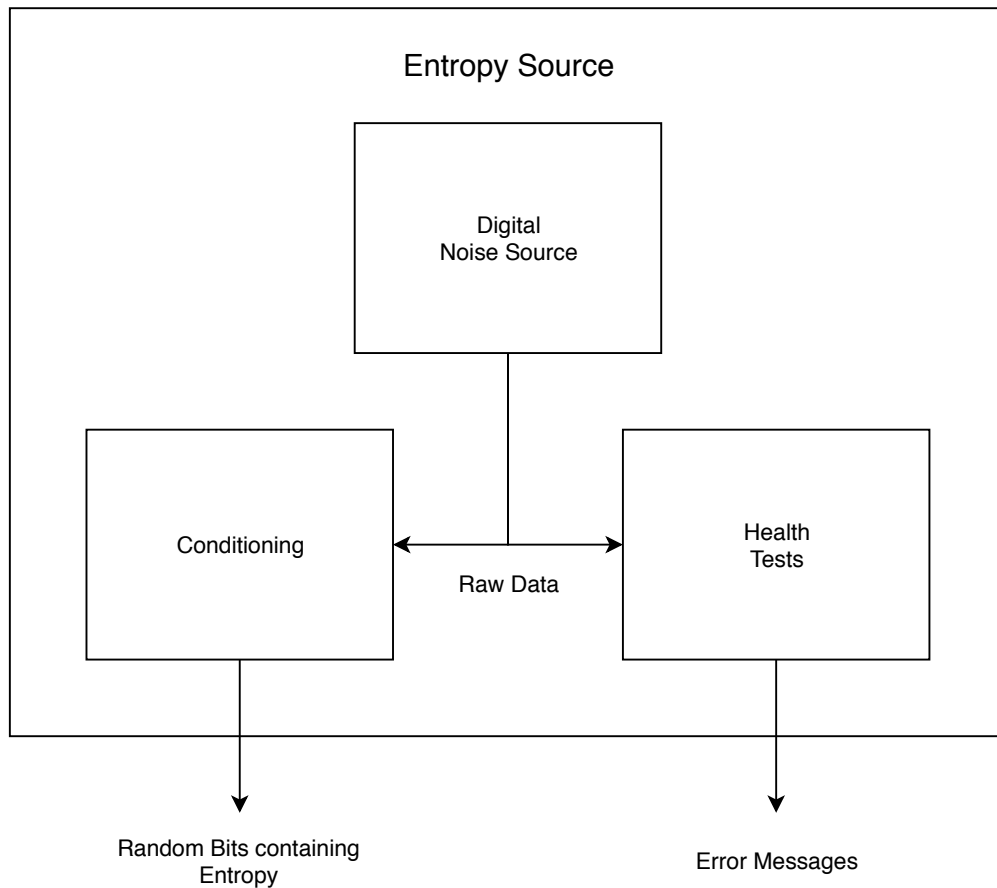


Figure 3: NIST SP 800-90B Entropy Source

## 3 Literature Review

The literature contains a plethora of information about randomness, entropy, and related concepts. It is easy to find (academic) articles about these topics, but harder to pick out the relevant ones. Additionally, there is no strong consensus a lot of the time, and there exist many competing theories. This section provides an overview over prior research which is of importance to this thesis.

### 3.1 Entropy

Entropy is the main concept underlying this thesis, which focuses on the interpretation of entropy being a property of randomness, measuring the amount of indeterminacy of the randomness. Initially, entropy was discovered in physics, specifically thermodynamics, and only later applied to Information Theory and cryptography, though the generalizations which have been made in these fields have been applied back to physics. Entropy appears to be a very general concept, applicable to many phenomena (section 2.1).

“Ueber verschiedene für die Anwendung bequeme Formen der Hauptgleichungen der mechanischen Wärmetheorie” by German physicist Rudolf Clausius [13] first introduced the concept of entropy in 1865, but is not really relevant to this thesis, as it is about thermodynamics, and deals with heat dissipation. It is still included here for completeness’ sake, as the article marks the very first discovery of the concept of entropy.

“Transmission of Information” by Ralph V. L. Hartley [29] laid out important groundwork for Information Theory and later interpretations of entropy in 1928, though the author did not realize this at the time. The article was written to find a quantitative measure which can be used to compare the capacities of different systems to transmit information. In retrospect, we can also interpret this as the amount of information that is revealed by observing a random process, which is what we now refer to as statistical entropy.

“A Mathematical Theory of Communication” by Claude E. Shannon [62] presents the first interpretation of entropy which is of importance to this thesis, and was discovered independently of the earlier notions of entropy. Even though the paper ([62]) builds on the aforementioned work from Ralph Hartley, neither Hartley himself nor Shannon realized that there was a connection to entropy. Only when talking about his discovery to mathematician and physicist John von Neumann, Shannon was told that entropy was an existing concept in physics, so he decided to stick with the name for his generalized version [67]. This also shed new light on Hartley’s article, and today we often refer to his function as Hartley entropy.

Building on the work of Claude Shannon, Alfréd Rényi generalized Hartley entropy (a.k.a. Max-entropy), Shannon entropy, Collision entropy, and Min-entropy into an encompassing description of entropy in his paper “On Measures of Entropy and Information” [57]. Rényi entropy  $H_\alpha$  is parametrized on order  $\alpha$ , where  $\alpha \geq 0 \wedge \alpha \neq 1$ . Different values of  $\alpha$  give different entropy measures, like  $\alpha = 0$ , which corresponds to Hartley entropy, or  $\alpha = 1$ , which corresponds to Shannon entropy. Higher values of  $\alpha$  result in more conservative entropy measures. Min-entropy gets its name from being the most conservative of the Rényi entropy measures, with the highest value of  $\alpha = \infty$ , and is used in this thesis.

## 3.2 Entropy Generation

As discussed in section 1, generating entropy using deterministic computers and algorithms is no easy task. On the other hand, due to its importance, it is also a well studied problem, and a lot of guidelines and best practices for entropy generation are available in the literature. This section contains a summary of this literature, insofar as it is relevant to this thesis.

Cryptographic hash functions in entropy generation are used for conditioning (section 5.3) and sometimes for output extraction. In his 1979 PhD thesis “Secrecy, Authentication, and Public Key Systems” [40], Ralph Charles Merkle laid out, among other things, a method of constructing collision resistant cryptographically secure hash functions, given a one-way compression function. Many widely used cryptographic hash functions have been derived by this method, including MD5, SHA1, and SHA2. This construction is known as *Merkle-Damgård construction* after Ralph Merkle and Ivan Damgård (see next paragraph), both of whom independently discovered the construction, and proved it to be sound. While this thesis does not go into the details of cryptographic hash functions, treating them as primitives, Merkle’s PhD thesis serves as a starting point for readers wanting to dive deeper into the matter.

Ivan Bjerre Damgård’s independent rediscovery of the *Merkle-Damgård construction* “A Design Principle for Hash Functions” [17] was published in 1989 in the proceedings of the *Conference on the Theory and Application of Cryptology*, and is included here for completeness’ sake.

Boaz Barak and Shai Halevi present an architecture for PRNGs, and a model thereof, in which they prove their architecture to be secure, in their article “A Model and Architecture for Pseudo-random Generation with Applications to `/dev/random`” from 2005 [7]. They consider *resilience*, *forward security*, and *backward security*, and advocate for the separation of the entropy extraction, and output generation, where the entropy extraction phase corresponds to entropy sources as described in section 2.3. Section 5 of their article is of special relevance to this thesis, and [7, section 5.3] contains a discussion of the relevance of their findings to `/dev/random` and `/dev/urandom`.

The American *National Institute of Standards and Technology* (NIST) “is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems” [68, p. i], and their *Information Technology Laboratory* (ITL) in collaboration with the *National Security Agency* (NSA) released a special publication (SP) titled “NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation” [68] in January 2018, which has already become an often cited source on entropy generation. Random Bit Generators (RBGs) as defined in their report are equivalent to RNGs in this thesis, as random numbers can be interpreted as bit-strings and vice versa. The NIST SP 800-90 series of reports contains SP 800-90A, which describes deterministic RBGs (equivalent to PRNG), SP 800-90B, which describes entropy sources, and SP 800-90C, with recommendations on how to combine these two. This structure is reminiscent of the approach by Barak and Halevi [7] (see above), who also separate entropy extraction from output generation. From the series, SP 800-90B is of highest pertinence to this thesis, and several definitions were adopted from it.

John Kelsey, one of the authors of NIST SP 800-90B, gave a presentation about it at the *Fault Diagnosis and Tolerance in Cryptography* conference titled “The 90B Approach to Entropy Sources” [37]. As its name suggests, it serves as a nice primer into the approach the ITL and NSA teams took to generate entropy, and a handy summary of SP 800-90B.

Germany’s counterpart to America’s ITL, the *Bundesamt für Sicherheit in der Informationstechnik*

(BSI) also conducted research into entropy and random number generation, and published their findings in their report “Evaluation of random number generators” [20] in 2013. Though I personally find it to be written in a needlessly incomprehensible style, it can still be a valuable resource. The cited document itself is merely a “master document referencing the current version of all documents” [20, p. 5] making up their methodology. NIST SP 800-90 is referenced by the BSI report, which was the predecessor to the NIST SP 800-90{A,B,C} series (page 12).

As discussed multiple times throughout this thesis, the Linux kernel (like all mainstream kernels) contains a cryptographically secure RNG, which is exposed to user space through the `/dev/random` and `/dev/urandom` devices, and since version 3.17 also as the `getrandom()` system call. On behalf of the BSI, Stephan Müller from *atsec information security GmbH* prepared the report “Documentation and Analysis of the Linux Random Number Generator” [41], which contains exactly what it says on the tin: analysis of and documentation for the Linux RNG, for every kernel version from 4.9 up to and including 4.20, which includes the latest stable version, as well as the 3 most recent long-term releases at the time of writing (2019-03-02) [66].

### 3.3 Entropy Attacks

“Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices” by Nadia Heninger et al. [32] demonstrates how entropy generation failures in the Linux kernel RNG can lead to exploitable weaknesses in cryptographic applications such as key generation. The article is discussed in more detail in section 4.2.

In 2014, Yevgeniy Dodis et al. extended the model by Barak and Halevi (section 3.2) in their article “Security Analysis of Pseudo-random Number Generators with Input: `/dev/random` is Not Robust” [18]. Their extension is concerned with recovering from state compromise, and they show that the architecture suggested by Barak and Halevi does not meet the extended requirements. Linux’s PRNG implementation is also evaluated according to the new model, and the findings show that it is not robust. This is demonstrated by two attacks, one on the entropy estimator, one on the mixing function. Finally, the authors suggest a new simple PRNG construction that can be proven robust in their model.

Marcella Hastings and Joshua Fried published their follow up paper to “Mining Your Ps and Qs” [32] titled “Weak Keys Remain Widespread in Network Devices” [30] in 2016, in which they collaborated with Nadia Heninger on measuring the response to her original 2012 paper. The authors found that most vendors did not appear to have produced patches for the original vulnerability, and that patches were mostly not applied by end-users where available. The number of vulnerable hosts actually increased after the public disclosure of the security weakness in 2012.

Brillo is an Internet of Things (IoT) operating system developed by Google, and based on the Linux kernel, thus inheriting its PRNG. Taeill Yoo et al. conducted an investigation into the security of this PRNG, which was released in 2017 under the title “Recoverable Random Numbers in an Internet of Things Operating System” [72]. They found that the underlying problems that lead to the vulnerability of the Linux kernel PRNG (section 4.2) were still present in Brillo.

Joel Gärtner’s 2017 master’s thesis “Analysis of Entropy Usage in Random Number Generators” [24] contains an investigation into how different PRNG implementations use entropy, and presents different approaches to entropy estimation. Gärtner also explains how entropy overestimation can lead to security vulnerabilities, and shows that there are scenarios in which real world entropy

estimators (e.g. in the Linux kernel) are too optimistic in their predictions, leading to outputs containing less entropy than desired and/or required.

Nadia Heninger, one of the authors of the aforementioned “Mining Your Ps and Qs” article as well as its 2016 follow up [30], gave a presentation titled “How not to generate random numbers” [31] in June of 2018. It covers several practical examples of random number generation having gone wrong, most of which were traced back to failures during entropy generation. The slides contain a lot of useful pointers to germane literature, some of which is covered explicitly in this thesis.

## 4 Examples

In the following, some examples of PRNG seeding failures are discussed.

Example 1 (section 4.1) covers a vulnerability that allowed stealing login cookies for a news site. It was chosen for disregarding every single best practice for entropy generation (section 5), as well as being easy to understand and explain.

Example 2 (section 4.2) is the Internet scale survey of vulnerable cryptographic keys performed by Nadia Heninger et al. [32] in 2012, in which they found a large number of RSA and DSA keys were vulnerable to compromise. The article is included in this thesis for being such a large scale survey, and for having been very influential. Additionally, as the authors note, their findings can be generalized to most mainstream OS kernels, so the article has high general applicability.

Example 3 (section 4.3) was included for being representative of a commonly used open source piece of software whose code has been public for years, yet it still contains obvious vulnerabilities in the currently distributed version (18.06 as of 2019-02-10).

Example 4 (section 4.4) is included as a representative for the booming gaming industry, and for being used in gambling software [23]. Like 7-zip, it has publicly available open source code, yet still contains a low quality entropy generator.

### 4.1 Hacker News Login Cookie Vulnerability

**Example 1.** American seed accelerator Y Combinator hosts a popular news forum called Hacker News. In 2009, user *dfranke* (David Franke) found a security vulnerability with regards to how the PRNG was seeded. The vulnerability drastically reduced the range of possible seed values. *dfranke* demonstrated that this could be used to guess login cookies by limiting the search space, which potentially allowed an attacker to perform account theft [21].

The PRNG which was used, among other things, to generate login cookies, used a seed based on the time the server was started. Here is the underlying code which was used to seed the RNG:

```
rs = scheme_make_random_state(scheme_get_milliseconds());
```

While it still requires the attacker to know the approximate startup time of the server to make brute forcing feasible, this is already a grave security vulnerability. It would have been possible to simply monitor the server for downtime, and record when it became available again.

This was not even necessary however, as *dfranke* also showed how this vulnerability could be paired with a thread exhaustion attack that forced a server restart. Using the combination of these two vulnerabilities reduced the search space of seeds on demand. The remaining entropy was low enough to comfortably allow for brute force attacks.

### 4.2 Widespread Weak Cryptographic Keys

**Example 2.** As discussed before, high quality randomness is required for cryptographic applications such as RSA and DSA. Their security models rely on the underlying RNG providing statistical quality randomness.



In their influential 2012 article “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices”, Nadia Heninger et al. [32] present the worrying results of their Internet scale network survey of cryptographic keys. They show that due to insufficient entropy during key generation, many TLS certificates share keys, or are susceptible to compromise. Even worse, the team was able to obtain RSA and DSA private keys for a number of TLS and SSH hosts.

The cause of the vulnerabilities has been traced back to the implementations, not the underlying cryptographic algorithms. Using “plausible software configurations” [32, p. 2], the team was able to reproduce the vulnerability, using the most common open-source software components from the population of vulnerable devices that was identified. RSA and DSA still appear to be secure when used in conjunction with correctly functioning cryptographically strong RNGs.

The authors note however, that “no one implementation is solely responsible” [32, p. 2]. Vulnerable keys were not limited to Linux, and were also observed under other operating systems like Windows and FreeBSD, but the team focused their etiology on Linux. They found that all tested implementations rely on `/dev/urandom`, which can provide deterministic output under certain conditions. When reading from `/dev/urandom` too early after booting up, it is possible for the entropy pool to not have been sufficiently initialized. `/dev/urandom` ignores the entropy estimate, as it is intended to be non-blocking, so it can silently provide insecure output. Embedded devices are especially likely to exhibit these conditions, due to their limited access to noise sources.

A better design would have been to never block after initially reaching the entropy threshold when booting up, but Linux has always had a strong focus on backwards compatibility, so changing the behavior in retrospect is seen as unacceptable by the developers [1]. What the Linux developers chose to do instead was to include a better interface to the existing PRNGs, via the system call `getrandom()` [26], which implements exactly this behavior. By default `getrandom()` extracts entropy from the same unblocking entropy pool as `/dev/urandom` (though this can be changed with flags), but instead of never blocking, it blocks until the pool has been fully initialized.

It is highly advisable to use this new interface over `/dev/urandom`, as it does not exhibit the boot time entropy vulnerability. `/dev/random` was never affected by the aforementioned vulnerability, but `getrandom()` still offers advantages, as it additionally addresses another vulnerability in which exhausting the file descriptors makes it impossible to open `/dev/random`, and forces the use of fallbacks, which are often less well tested, and less secure [1].

### 4.3 Badly Implemented Random Number Generator in 7-Zip

**Example 3.** On 22nd of January 2019, Twitter user “@3lbios” (Michal Stanek) posted a series of Tweets on Twitter [63], detailing a number of security vulnerabilities he found in 7-Zip’s cryptography code.

Instead of using a proper cryptographically secure PRNG based on a peer reviewed algorithm, 7-Zip includes a very weak ad hoc implementation of a PRNG. Apparently the author was well aware of the weaknesses of this generator. Included with the implementation are comments explaining the shortcomings, and asking the reader to only use it for salting.

Listing 1: 7-Zip 18.06 PRNG (`RandGen.cpp`) [2]

```
1 #include "StdAfx.h"
2 #include "RandGen.h"
```

```

3
4 #include <time.h>
5
6 // This is not very good random number generator.
7 // Please use it only for salt.
8 // First generated data block depends from timer and processID.
9 // Other generated data blocks depend from previous state
10 // Maybe it's possible to restore original timer value from generated
   value.
11
12 #define HASH_UPD(x) Sha256_Update(&hash, (const Byte *)&x, sizeof(x));
13
14 void CRandomGenerator::Init() {
15     CSha256 hash;
16     Sha256_Init(&hash);
17
18     pid_t pid = getpid();
19     HASH_UPD(pid);
20     pid = getppid();
21     HASH_UPD(pid);
22
23     for (unsigned i = 0; i < 1000; i++) {
24         time_t v2 = time(NULL);
25         HASH_UPD(v2);
26
27         for (unsigned j = 0; j < 100; j++) {
28             Sha256_Final(&hash, __buff);
29             Sha256_Init(&hash);
30             Sha256_Update(&hash, __buff, SHA256_DIGEST_SIZE);
31         }
32     }
33     Sha256_Final(&hash, __buff);
34     __needInit = false;
35 }
36
37 void CRandomGenerator::Generate(Byte *data, unsigned size) {
38     if (__needInit)
39         Init();
40     while (size != 0) {
41         CSha256 hash;
42
43         Sha256_Init(&hash);
44         Sha256_Update(&hash, __buff, SHA256_DIGEST_SIZE);
45         Sha256_Final(&hash, __buff);
46
47         Sha256_Init(&hash);

```

```

48     UInt32 salt = 0xF672ABD1;
49     HASH_UPD(salt);
50     Sha256_Update(&hash, _buff, SHA256_DIGEST_SIZE);
51     Byte buff[SHA256_DIGEST_SIZE];
52     Sha256_Final(&hash, buff);
53     for (unsigned i = 0; i < SHA256_DIGEST_SIZE && size != 0; i++,
          size--)
54         *data++ = buff[i];
55     }
56 }
57
58 CRandomGenerator g_RandomGenerator;

```

The PRNG is seeded by using the process ID (`getpid()`), the parent's process ID (`getppid()`), and the current time (`time(NULL)`). All of these noise sources are very predictable, which means they are not suited for a cryptographic entropy source. Also, because `time()` has a 1 second resolution, it is very likely that `v2` will share the same value across a significant amount of the 1000 iterations. While SHA256 is used for conditioning (section 5.3), this does not fix the underlying issue.

During entropy generation, a fixed salt (`0xF672ABD1`) is mixed into the entropy pool. This is obviously very bad, as a fixed value does not add any entropy. Combined with previous state, which is also deterministic, this is the only data influencing the output. In essence, that means no new entropy is ever mixed into the pool, making the RNG highly insecure.

Blatantly disregarding the warnings about its insecurity, the ad hoc PRNG implementation is called from security critical code, where it is used to generate Initialization Vectors (IVs) for AES encryption [2]. The purpose of IVs is to prevent repetition during the encryption, hindering dictionary attacks. This turn of events showcases a frequent phenomenon: People are lazy. The author of the RNG code was lazy for not implementing a proper noise source, and for using an unproven ad-hoc RNG design. The person who used this insecure PRNG in security critical cryptographic code was lazy for not using a better RNG, and/or for not looking at the comments warning about the shortcomings of the included RNG. This is not intended as an attack on the 7-zip authors. Laziness is a natural tendency in many animals, including humans. Rather, it emphasizes the importance of improving the state of the art of entropy generation. As long as there are insecure PRNGs, they will inevitably be misused in security critical applications.

As of version 19.00, 7-Zip now reads from `/dev/urandom` [3], which is definitely an improvement over version 18.06, but still not ideal. A better approach would be using the `getrandom()` system call (see section 4.2), which is actually included, but commented out for some reason.

7-Zip also contains an even worse implementation of a PRNG, which uses only a single noise source, and does not perform any conditioning. In addition, `rand()` is typically not cryptographically secure. This PRNG is not used for any security critical tasks though, only to create some names [2]. Due to this, the second PRNG is of no direct concern, but included here anyway, because it is indicative of common anti-patterns.

Listing 2: 7-Zip 18.06 PRNG (Random.cpp) [2]

```

1 #include "StdAfx.h"
2 #include "Random.h"
3
4 #include <stdlib.h>
5 #include <time.h>
6
7 void CRandom::Init(unsigned int seed) { srand(seed); }
8
9 void CRandom::Init() { Init((unsigned int) time(NULL)); }
10
11 int CRandom::Generate() const { return rand(); }

```

## 4.4 Unsuitable Noise Source in the Godot Game Engine

**Example 4.** Godot [27] is a free and open-source game engine, licensed under the permissive MIT license. Combined with its expansive feature set, including but not limited to state of the art 3D rendering, a dedicated 2D engine, and cross platform support, this makes it a popular choice for developing independent computer games.

Many games require a source of randomness, and therefore – if implemented in software – a (P)RNG [24]. Just think of any card game like the many variants of Poker, or gambling games like Roulette, or slot machines. These games belong to a large category of games relying in full or part on chance.

Godot uses PCG32, a specimen of the PCG family [43] of random number generators with a 64-bit internal state and 32-bit output. In and of itself, members of the PCG family are cryptographically secure, but depend on a seed like all PRNGs. However, the way this seed is chosen by the Godot engine negates all security guarantees made by PCG:

Listing 3: Godot RNG (excerpt from random\_pcg.cpp) [28]

```

41 // ...
42
43 void RandomPCG::randomize() {
44     seed(OS::get_singleton()->get_ticks_usec() * pcg.state +
45         1442695040888963407ULL);
46 }
47 // ...

```

`randomize()` (listing 3) is the function which is used to set the seed for the Godot RNG. It uses a linear congruential generator where the multiplier is a timestamp.

Listing 4: Godot RNG (excerpt from `random_pcg.h`) [28]

```

47 // ...
48
49 void seed(uint64_t p_seed) {
50     current_seed = p_seed;
51     pcg.state = p_seed;
52     pcg32_random_r(&pcg); // Force changing internal state to avoid
        initial 0
53 }
54
55 // ...

```

`seed()` (listing 4) simply sets the state to the given seed, and additionally skips one value.

Listing 5: Godot RNG (`pcg.cpp`) [28]

```

1  // *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.
    org
2  // Licensed under Apache License 2.0 (NO WARRANTY, etc. see website)
3
4  #include "pcg.h"
5
6  uint32_t pcg32_random_r(pcg32_random_t* rng)
7  {
8      uint64_t oldstate = rng->state;
9      // Advance internal state
10     rng->state = oldstate * 6364136223846793005ULL + (rng->inc|1);
11     // Calculate output function (XSH RR), uses old state for max ILP
12     uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
13     uint32_t rot = oldstate >> 59u;
14     return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
15 }

```

`pcg32_random_r()` (listing 5) is just a standard implementation of PCG32.

The only noise source which introduces entropy into the Godot’s PCG state is the timestamp. It does not contain sufficient entropy in order to secure the seed from being compromised (section 5.1).

It is not easy to use different noise sources within the Godot engine, so it stands to reason that most game developers will just use what is available, and not bother implementing a better noise source. This is especially problematic because Godot is used for developing gambling games, where the quality of randomness used by a game influences the fairness (or lack thereof) of this game. Additionally the prevalence of e-sports has surged in the past few years, and will probably continue to do so. With tournaments regularly attracting huge crowds, and prize money in the millions, a great incentive is created for exploiting every method of gaining an advantage over adversary players, including using weaknesses in a game’s RNG. All of this means that there can be real

money on the line, and high quality randomness is essential to prevent highly motivated individuals from exploiting weaknesses in games.

Ideally, a game engine like Godot should facilitate the generation of high entropy cryptographically secure randomness, especially considering that every modern mainstream operating system provides a cryptographic RNG, so the responsibility of generating better seeds could be foisted on the host OS. In a perfect world, the path of least resistance is also the best one, so Godot should make using cryptographically secure randomness easier than using low entropy randomness, which is the only way to reliably prevent developers from implementing highly insecure games. If making a game insecure takes more effort than making it secure, developers will start implementing secure games.

The programmers at Gamblify [23], a Godot sponsor and independent Danish producer of hardware and software for the gambling industry (e.g. physical slot machines and online casino software), have recognized and worked around the insufficiency of Godot’s noise source. They only use Godot for the user interface of their slot machines, while the game logic itself resides within the backend which includes a Java PRNG (`java.util.Random`) seeded by an entropy source implemented in C++ (`std::random_device`). This approach required implementing a custom TCP based integration with Godot, in order to be able to send the results from the backend to the Godot game, which is a nontrivial endeavor [49].

In conclusion, the default method provided for seeding the PRNG included with Godot is insecure. Obtaining proper cryptographically secure randomness requires more effort than should be necessary, which nudges game developers into the wrong direction, because many if not most of them will take the path of least resistance, which in the case of Godot is the insecure path. Working around this limitation takes considerable effort, and will not be undertaken by a majority of Godot users.

## 5 Entropy Generation Best Practices

In this section I will describe the best practices for entropy generation, as well as common pitfalls.

### 5.1 High Entropy Noise Sources

Noise sources are the core of every entropy source, which are themselves integral to PRNGs. They should contain inherently high entropy, which roughly means that no strategy of predicting the output of a noise source should give a potential attacker an advantage over picking at random. Predictable noise sources can completely compromise the security of anything relying on a PRNG seeded by randomness derived from them (section 2.2).

Entropy can be seen as the amount of indeterminacy of some variable. If a fair coin is flipped, the outcome can be represented by 1 bit, and the outcome is completely indeterminate. No method of guessing the outcome will result in a better average rate of correct guesses than  $\frac{1}{\text{number of outcomes}} = \frac{1}{2}$ . Therefore, each coin flip contains 1 bit of entropy (section 2.1).

Using a simple example, the vulnerability caused by predictable noise sources can be demonstrated. On the one hand, we have a program which uses a very predictable noise source: the current time. It uses this noise source directly as a seed to the C standard library random number generator. After printing the seed to `stdout`, the program generates `NSAMPLES` (defined in listing 33; 10 per default) random number samples, and writes them to an output file (listing 6).

Listing 6: Generator (`poc.c`)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include "util.h"
6
7 int main() {
8     time_t seed = time(NULL);
9     printf("seed: %ld\n", seed);
10    srand(seed);
11    FILE *output_file = open_file("out.txt", "w");
12    for (int i = 0; i < NSAMPLES; i++) {
13        fprintf(output_file, "%d\n", rand());
14    }
15    fclose(output_file);
16    return EXIT_SUCCESS;
17 }
```

On the other hand, we have an attacker program (listing 7). It reads the output file, and guesses the seed based on the modification time of the file. Starting with a seed value slightly below the modification time (i.e. when the generator program has likely been run), the attacker increases the seed until the generated samples match the samples from the output file. After being able to reproduce all `NSAMPLES` random samples, the likely seed is printed to `stdout`.

Listing 7: Attacker (attack.c)

```

1 #include <errno.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #include "util.h"
6
7 /**
8  * Guess the seed by comparing the samples.
9  *
10 * @return guess of the seed that produced the samples
11 */
12 unsigned int guess_seed(unsigned int seed, int samples[]) {
13     for (;;) seed++;
14     srand(seed);
15     for (int i = 0; i < NSAMPLES; i++) {
16         if (i == NSAMPLES) { return seed; }
17         if (rand() != samples[i]) { break; }
18     }
19 }
20
21
22 int main(int argc, char *argv[]) {
23     if (argc != 2) {
24         fprintf(stderr, "Exactly one command line argument required\n");
25         return EXIT_FAILURE;
26     }
27     int samples[NSAMPLES];
28     FILE *input_file = open_file(argv[1], "r");
29     for (int i = 0; i < NSAMPLES; i++) {
30         fscanf(input_file, "%d\n", samples + i);
31     }
32     printf("seed: %u\n", guess_seed(modification_time(argv[1]) - 1,
33                                     samples));
34     fclose(input_file);
35     return EXIT_SUCCESS;
36 }

```

After running the first program, the attacker program can recover the seed, just from looking at the output file:

```

$ ./poc
seed: 1548979324
$ ./attack out.txt
seed: 1548979324

```

Guessing the seed is straightforward, because the indeterminacy of the noise source is extremely



low. Limiting the range of possible seed values enough to make brute forcing feasible is facile, because we can use the modification time of the output file to estimate the time at which the program was run. This modification time is stored in the file meta data on most popular file systems.

Even without this luxury, we could still very easily constrain the search space. If we knew, the program has been run within the last ten days, for example, that would equate to roughly 1 million possible seed values. Checking all of them takes less than a second, using the code above.

It might seem like this is an unrealistic scenario, but examples like this are actually very common, and easy to find (section 6.1). A situation very similar to this can even occur in the Linux RNG, in circumstances with limited noise sources. Nadia Heninger et al. even observed servers in the wild that apparently had no noise source whatsoever [32].

## 5.2 Multiple Noise Sources

We define a noise source as some source of inherently random bit strings. A noise source could be a physical process like radioactive decay, background radiation, mouse movement, or any other sensor input that contains randomness. These processes can be used as the basis of an entropy source [37].

A noise source is an integral part of any random number generator. Without an inherently random noise source, we cannot generate entropy. The PRNGs are deterministic, and so the randomness has to come from the seed. The seed is generated from an entropy generator, which is itself deterministic, so its randomness has to come from the noise source [37, 68].

If only a single noise source is used, an attacker who is able to control or intercept this source has immediately compromised our system. Ideally, the entropy generator would catch a compromised noise source by using health tests (section 5.4) on the raw input [31, 37, 68]. However, entropy estimation is really hard in practice, and most entropy estimators are relatively easy to fool. Even if we had a perfect entropy estimator, the attacker could intercept the random bits, look at them, and pass them on to the entropy generator without any modification. That way even perfect randomness does not guarantee security, if the random stream is not kept confidential.

Therefore, it is advisable to use multiple noise sources. An attacker would need to control or intercept every randomness source in order to compromise the entropy generator. For this to work, we need to mix in sufficient entropy from each noise source for the conditioning, otherwise compromising some of the randomness sources could enable brute force attacks by limiting the search space. Provided that enough random bits from each of the noise sources are used, brute forcing the input noise is still at least as hard as brute forcing the output from the entropy generator itself, making this attack obsolete.

## 5.3 Conditioning

The noise sources might be random, but not evenly distributed. To rectify this, the input noise should be uniformly redistributed. We call this process “conditioning” [31, 37, 68].

It is important to note that conditioning is an optional step when implementing an entropy source. Conditioning can increase the entropy per bit by redistributing the biased output of a noise source, but this only works if there is still some inherent randomness in the noise source. 1000 consecutive

timestamps do not provide more entropy compared to a single timestamp. Conditioning can “concentrate” entropy in a lower number of bits to meet some entropy target, but does **not** add entropy itself.

Cryptographic hash functions like SHA-256 can fill the role of a conditioner in an entropy generator [70, 37, 68]. The SHA-256 turns an input bit sequence of arbitrary length into a 256-bit, evenly distributed output bit string. Cryptographic hash functions possess the properties of collision resistance, pre-image resistance, and second pre-image resistance. Also a small (e.g. 1 bit) change in the input should lead to a large change in the output. These properties make cryptographic hash functions suitable as conditioners in entropy generators.

There also exist non-cryptographic conditioning algorithms that can be used to redistribute the non-uniform output from a noise source. Von Neumann unbiasing is an example of such an algorithm [37, 42]. It relies on the fact that regardless of the probability of 0 or 1 occurring in the output, the probability of 01 occurring must be the same as 10. Von Neumann suggested rejecting occurrences of 00 and 11, and using the first bit from 01 and 10 respectively. This way the input stream is redistributed uniformly, no matter what the underlying distribution of 0s and 1s is [42].

Von Neumann unbiasing relies on neither the probability of 0, nor the probability of 1 occurring being 0. This assumption must hold for noise sourced anyway, otherwise they cannot provide any entropy. It also relies on a constant probability distribution of 0s and 1s, as well as individual bits being independent. This is somewhat harder to prove for noise sources in practice.

The output of the conditioning step can either be used directly as the output of the entropy generator, or processed further, for example using a stream cipher [70].

## 5.4 Health Testing

As discussed earlier, an entropy generator should try to detect major failures in randomness generation from the noise source. If an attacker assumes control of a noise source, or if the noise source starts giving significantly non-random output for some other reason, the entropy generator should detect this. If the health checks fail, the entropy generator should refuse to provide output, and instead give some error message, or enter some failure state [31, 37, 68].

By definition, entropy estimation is really difficult, and even theoretically impossible. Entropy cannot be measured directly, as that would be a contradiction, given the definition of entropy as “the amount of indeterminacy”. If we could mathematically define a sequence as random, it would no longer be random [22, p. 9–10]. Another interpretation is that no absolute randomness exists, only randomness with respect to certain properties [64, p. 4–5]. Entropy estimation being difficult is no reason not to try to do as best as we can, however.

There are different ways of going about estimating entropy. Entropy estimators can either collect some input noise, and perform statistical randomness tests on it, or try to indirectly estimate entropy continuously while it is being received. `/dev/random`, for example, takes the latter approach, and has been criticized for doing so [7, 72]. In essence, the Linux RNG estimated incoming entropy based on its timing, not based on the data itself. Therefore very low entropy data with irregular timing can be estimated to contain higher entropy compared to a source of high-quality entropy that provides samples regularly.

However, the former approach can be relatively easily tricked as well, even ignoring the questionable

validity of talking about the entropy of random sequences as opposed to random processes (section 2.1). A potential attacker could take a sequence of inputs, which contains no entropy from his point of view, for example a sequence of consecutive integers. By running this sequence through a cryptographic hash function, he could turn it into a seemingly random sequence from the user's point of view. An entropy estimator would classify this sequence as containing high entropy due to the properties of cryptographic hash functions, even though it does not for the attacker.

Health testing should not be taken as infallible, or seen as a reliable protection against attacks, but more as a sanity check protecting us from obvious failures of our noise sources. Imagine, for example, some accelerometer being used as a noise source suddenly getting disconnected. Accelerometers can be used as viable noise sources providing high entropy even in a resting state [70], and no attacker is present. In this scenario, the health check should catch that the input from the accelerometer sensor is suddenly very low entropy, and refuse to provide further output.

## 5.5 Advice for Implementors

Everybody finding themselves in the position of needing to generate random numbers on a deterministic computer for some reason or another should follow the advice from this section, unless they know exactly what they are doing, and why they are deviating from said advice.

- **Use a cryptographically secure PRNG.** It might not seem obvious why a specific use case might benefit from this property, but there is also no disadvantage. k-dimensionally equidistributed cryptographically secure random number generators with arbitrary periods exist, which exhibit very favorable space and time performance, are difficult to predict, and possess additional useful features like multiple streams, and compact code size [43]. They are even suitable for use in embedded devices, though entropy generation in these environments remains a problem (section 4.2). If you are not sure which PRNG algorithm to use, use PCG32, which has easy to integrate C, C++, and Haskell implementations [48].

Most (security) vulnerabilities are obvious in hindsight, but they still occur all the time. In many cases, implementors simply did not think about how some features could be exploited, or it was not obvious how this could be done. In other cases, it is assumed that everybody ever working with the code will pay close attention to using the correct PRNG implementation in the correct place, which is an unrealistic standard to keep up, and entirely unnecessary. Vulnerabilities often arise from complex interactions between different parts of a codebase, which are non-trivial to understand, and it cannot be expected that they will always be considered in their entirety.

Using an insecure RNG for a security critical task can have calamitous consequences, but using a cryptographically secure RNG for mundane tasks is harmless. If a project does not make use of non-cryptographic PRNGs, programmers can rest assured that no unforeseen interactions between different systems can expose exploitable vulnerabilities. The 7-zip example (section 4.3) shows that even explicit comments warning about insecure PRNGs and seeds do not reliably stop them from being used for security critical applications. Banishing insecure PRNG implementations from codebases can prevent a whole class of security vulnerabilities, without any drawbacks. At some point this might have been prohibitively expensive, as cryptographically secure PRNGs have traditionally been more resource intensive, but the state of the art of pseudo random number generation has advanced since those days.

- **Let the operating system (OS) generate seeds for you.** As extensively discussed throughout the whole thesis, generating cryptographically secure seeds on deterministic hardware is difficult and error prone, so do not try to do it yourself if you can avoid it. All mainstream operating systems provide an interface to a cryptographically secure random number generator. These implementations have been intensely scrutinized, and continually improved over the lifetime of their OS, and a lot of time and effort has been put into trying to increase their security, due to their central role in providing cryptographic security (sections 3.2, and 3.3). Most implementors will not be able to put a comparable amount of due diligence into constructing their own entropy generators (and indeed, most do not even try; see sections 4, and 7.4), so their first impulse should always be to choose an existing proven implementation. Operating systems are uniquely suited for entropy generation, because they have a deeper and more comprehensive overview over the system compared with user space processes, which makes collecting noise easier [70].

When using the secure random number generation facilities provided by the OS, make sure to use an appropriate interface, for example the `getrandom()` system call instead of `/dev/urandom` on Linux (section 4.2) [1]. Of course, you should also take care to treat your seeds confidentially at all times.

- **If you absolutely have to implement an entropy source yourself, follow the best practices.** This may be seemingly obvious advice, but experience shows that implementors often fail to take even the most basic best practices into consideration when implementing entropy sources (see sections 6, and 7.4). The best practices are outlined in this section, as well as sections 2.2 and 2.3. Therefore, the following is intended only as a short summary.
  - Pick your noise sources carefully (section 5.1), and try to understand their behavior and failure states as best as you can. You should be able to explain where the entropy in your noise source comes from, and give well-founded estimates for the amount of entropy they output per bit. Additionally, you should be able to account for the failure states of your noise sources, and develop techniques to reliably detect them.
  - Whenever you can, use multiple noise sources (section 5.2), but not at the expense of the quality of individual sources. A single high quality noise source is preferable over combining several low quality ones. If you combine multiple noise sources, take sufficient entropy from each source, so that even knowing the output of all other sources, each single sources provides security on its own, requiring a potential attacker to observe or take control over every single noise source in order to compromise your entropy generator.
  - Perform health testing (section 5.4), by monitor your entropy source for the failure states you have isolated, and ideally for any unexpected suspicious activity which might not have been anticipated. As soon as some failure state is detected, refuse to provide any further entropy output, and raise an error instead.
  - If you fall short of your entropy targets, use conditioning (section 5.3) to “condense” the entropy into a smaller number of bits.

## 6 Entropy Generation in Practice

Evidently, implementing a high quality entropy source is still a difficult endeavor. This section is about how entropy generation can go wrong in practice. In contrast to the previous section, it specifically covers cases in which the best practices were not followed.

	Predictable Noise Source(s)	Single Noise Source	No Conditioning	Missing Health Tests
Hacker News Login Cookie (section 4.1)	✓	✓	✓	✓
Weak Cryptographic Keys (section 4.2)	✓	✓ <sup>(1)</sup>		✓ <sup>(2)</sup>
7-Zip Weak RNG (section 4.3)	✓	✓ <sup>(3)</sup>		✓
Godot Unsuitable Noise Source (section 4.4)	✓	✓		✓

Figure 4: Overview of Entropy Generation Weaknesses

- (1) While Linux tried to gather entropy from multiple noise sources into an entropy pool, in some situations only a very limited number of noise sources were available. Nadia Heninger et al. [32] were able to reproduce realistic scenarios in which only a single noise source was available, or even no noise source at all.
- (2) Linux has an entropy estimator which tries to estimate the entropy of the input pool through the timing of events it uses as noise sources. Regardless of the questionable validity of this approach, the entropy estimates are completely ignored when reading from `/dev/urandom`. Following the definition laid out in section (section 2.3), `/dev/urandom` does not practice proper health testing, because it should refuse to give output when its entropy estimate is too low. This runs contrary to the express purpose of `/dev/urandom` as a non-blocking alternative to `/dev/random`.
- (3) 7-zip uses multiple noise sources: the pid of the 7-zip process, the pid of the parent of the 7-zip process, and multiple timestamps (though they are very likely to significantly overlap (section 4.3)). Strictly speaking it therefore does fulfill this requirement, but before using multiple noise sources one should make sure that at least one of the noise sources provides sufficient entropy on its own, which is not the case for 7-zip, as explained in sections 5.2 and 6.2.

## 6.1 Predictable Noise Sources

Noise sources are required to be inherently random (section 2.2). If an attacker can predict the output of a source, it is unsuitable for use as a noise source in an entropy generator.

A common example of an unsuitable noise source is the current time. It is frequently used due to the simplicity of retrieving it, and because it changes every time the program is run. On cursory inspection, things seem to be fine, as the output value changes every time. That does not mean that the noise source contains high entropy however, as demonstrated by the proof of concept in section 5.1, and the work by Nadia Heninger et al. [32].

All of the examples (see section 4 and figure 4) used the current time as a source of entropy, which demonstrates how common this pattern is. These examples were not specially picked in any way, but when entropy generation fails, often timestamps seems to be involved. Other common bad noise sources include process identification numbers (pids), and clock ticks. Note that all of these common unsuitable noise sources are directly (timestamps, clock ticks) or indirectly (pids) related to the process starting time, so combining them does not increase the entropy in the entropy pool by the amount one would expect if the sources were independent.

Bad noise sources can contain some limited entropy from the point of view of an attacker, but do not add much in terms of security. Mixing a timestamp, pid etc. into the entropy pool is not intrinsically bad in the case of multiple noise sources, but this practice becomes problematic when they are used as the only noise source, or together with other predictable noise sources (section 6.2).

## 6.2 Single Noise Source

In case of some noise source providing less entropy than predicted, or in case of it being controlled or observed by an attacker, the best defense is having other noise sources which can compensate for this lack of entropy. Ideally, but not necessarily, each noise source should provide sufficient entropy on its own to make brute forcing infeasible. This way a potential attacker would need to seize control of or observe every single noise source in order to compromise the system (section 5.2).

Like with predictable noise sources, none of the examples followed all best practices for using multiple noise sources (figure 4). Linux's RNG tries to use multiple noise sources, but sometimes they are not available, 7-zip uses multiple noise sources, but each of them is insufficient, and finally Hacker News and Godot outright only use a single noise source.

Using a low entropy source, and using a single noise source are problems that mutually exacerbate each other. Having a single noise source is not so bad, if it is a high entropy noise source, and having low entropy noise sources can be counteracted by using multiple of them. In practice however, low entropy noise sources are often used as the sole noise source, or only combined with a small number of other low entropy noise sources.

## 6.3 No Conditioning

Conditioning is optional when implementing an entropy source, yet it can still be very useful in order to redistribute the noise evenly, and increase the entropy per bit (section 5.3).

Looking at the vulnerability overview (figure 4), we see that most examples applied some form of conditioning. Linux’s RNG and 7-zip use cryptographic hash functions for conditioning, while Godot uses a linear congruential generator. The only example in which no conditioning was used is the Hacker News vulnerability.

All of the examples still exhibited some vulnerability though, which demonstrates how conditioning cannot fix failures that arise earlier in the entropy generation process. These failures can be obfuscated by conditioning, which might be a disadvantage in some cases, because it can lead to delays in the discovery and fixing of the underlying problems. Well functioning health tests can help to alleviate this drawback (section 5.4), but they were also absent in most examples (section 6.4).

## 6.4 Missing Health Tests

Health tests should have recognized the predictability of the unsuitable noise sources used in all of the examples, and output an error leading to the entropy sources refusing to provide further output (section 5.4).

Linux’s RNG implementation is the only example which incorporated health tests into its design, though they are completely ignored when reading from `/dev/urandom`, which obviously makes them useless in this case. None of the other examples included health tests (figure 4).

While it is hard to speak to the intentions of the authors when writing the code, comments left by the implementors can be combined with other indicators to make reasonable guesses about why they made the decisions they did. After inspecting the code, it seems that often times the programmers were aware of the low quality of their noise sources, and intentionally excluded health testing. When the noise source inherently provides low entropy, then health tests must either be of low quality in order to not catch these obvious failures, or they will constantly fail and prevent output from being generated. Health tests abiding by the former behavior are superfluous, so it makes sense not to include them at all. From a security point of view, the latter behavior is preferred, but from a practical point of view, it would prevent the RNG from ever functioning. In practice most implementors apparently prefer a working insecure RNG over a secure one which refuses to provide output.

For the 7-zip example, reconstructing the implementor’s thought process is the easiest, because the code (listing 1) explicitly contains comments explaining how the RNG is really insecure, and should only be used to generate salts. The author even warns about the possibility of compromising the seed just from looking at the output of the RNG. Because the RNG was never intended to be used for security critical tasks, it is understandable that the author chose to eschew health testing.

When a RNG was needed, but it was not implemented yet, the author reached for (what he thought to be) the easiest tool available: an insecure ad hoc implementation. Later, when a RNG was needed in another part of the codebase, the author – again – chose the path of least resistance and used the implementation which was already available. Both decisions are understandable in isolation, but detrimental in combination. Corners will inevitably be cut, so we should aim for purging insecure random number generation in general. If there are no insecure PRNGs, they cannot be abused for security critical code.

With Linux, the reasoning behind ignoring the health tests when reading from `/dev/urandom` is also pretty obvious. `/dev/urandom` is specifically intended as a non-blocking alternative to

`/dev/random`. As noted by Theodore Ts'o [1], the focus has been on trying to initialize the entropy pool as quickly as possible, before it can be read from. This is an admirable goal, but not always achievable. A better design, as well as the solution actually implemented by the Linux developers are discussed in section 4.2.



## 7 Detecting Entropy Generation Weaknesses

As demonstrated in this thesis, knowing about best practices and possible vulnerabilities is no sufficient defense. Each implementor will have the tendency to cut as many corners as possible. Decisions that are harmless in isolation can become security vulnerabilities when interacting with other seemingly benign decisions (section 4.3). The most effective way to improve the state of entropy generation in practice is automation. Errors are unavoidable, and should ideally be caught as early as possible by automated code review tools.

LGTM [4] is such a code review system, which can be used to automatically find anti-patterns and vulnerabilities, based on the same observation made in this thesis: the same bug often surfaces multiple times, across time and project boundaries. Using LGTM, bugs can be categorized into so called *variants* to find similar problems, in a process called *variant analysis*. Whole classes of vulnerabilities can be eliminated by this process, and queries can be shared with other developers and researchers, thus the gained expertise can proliferate easily.

QL [5] is the query language driving this variant analysis. It is an object-oriented logic programming language semantically similar to SQL, and is used to query databases which LGTM generates from a project's code. While its syntax is based on SQL, its semantics are based on the declarative logic programming language Datalog, which is itself based on Prolog, which implements propositional logic. QL allows us to write queries like the following:

Listing 8: QL query: Pythagorean triples

```
1 from int x, int y, int z
2 where x in [1..10] and y in [1..10] and z in [1..10]
3     and x*x + y*y = z*z
4 select x, y, z
```

The query (listing 8) was taken from the official introduction to QL [36], and computes all Pythagorean triples ( $x^2 + y^2 = z^2$ , where  $x, z, y \in \mathbb{Z}^+$ ), where  $x, y, z \in [1, 10]$ . Making use of the object-orientation in QL, we can rewrite the query as follows:

Listing 9: QL query: Pythagorean triples with object-orientation

```
1 class SmallInt extends int {
2     SmallInt() { this in [1..10] }
3     int squared() { result = this.pow(2) }
4 }
5
6 from SmallInt x, SmallInt y, SmallInt z
7 where x.squared() + y.squared() = z.squared()
8 select x, y, z
```

QL also offers more advanced features like aggregate functions (e.g. `max()`, `count()`), quantifiers (e.g. `forall`, `exists`), and recursion. These features allow users to succinctly express complex queries like the following:

Listing 10: QL query: longest function name (C/C++)

```
1 import cpp
2
3 from Function f
4 where not exists(Function g |
5     g.getName().length() > f.getName().length()
6 )
7 select f.getName(), f.getName().length()
```

The above QL code (listing 10) queries a C or C++ codebase for the function with the longest name. Running it against the Linux kernel reveals that there are two functions which both have a name which is 66 characters long, and both are related to the ext4 file system.

The remainder of this section is dedicated to finding entropy generation weaknesses using QL queries in LGTM.

## 7.1 Database extraction

LGTM provides support for querying C/C++, C#, COBOL, Java, JavaScript, and Python codebases through QL libraries. These libraries provide the framework on which the analysis is based [4]. In the example query above (listing 10), we imported the `cpp` library, which is used for C and C++, because we wanted to analyze the Linux kernel's C source code.

Before a codebase can be queried, a database needs to be generated from it, which is what the QL queries then run against [25]. In fact, a database is generated for each commit to track changes over time, but that is secondary to understanding how this works. Each database contains an abstract syntax tree (AST) representing the codebase, and additionally some supplementary information. The database extraction step allows for a unified representation of codebases using different programming languages, as well as efficient querying.

Programming languages are extremely diverse, so every supported language comes with its own extractor, which is responsible for turning the codebase into a queryable database, and its own database schema with a table representing each language constructs. Using multiple custom extractors, as opposed to a universal representation, ensures that the analysis can be as accurate as possible.

To be able to build a database, LGTM has to determine which files to analyze. For interpreted languages such as JavaScript and Python, the extractor runs directly on the source, and resolves dependencies. For compiled languages (e.g. Java, or C/C++) on the other hand, it works by observing the build process [25]. Of course, that means that LGTM needs to be able to build the project, which may require custom configuration. The Linux kernel is included as an example with LGTM, and was already imported, so we did not need to worry about this, but if we want to import new projects, this becomes relevant. LGTM comes with default configuration for every supported language, but oftentimes this is not sufficient. In this case, an `lgtm.yml` or `.lgtm.yml` configuration file is required to be in the repository, in order to customize the build process to suit the project at hand [38].

The extraction process consists of multiple stages which can all be individually configured [14]:

1. **prepare:** In the preparation step, dependencies are installed, and necessary folders for the analysis are created. This stage does nothing by default.
2. **after\_prepare:** This stage can be used, for example, to clean up things from the preparation step, or for setting up things (e.g. environment variables) for later stages, etc. This stage does nothing by default.
3. **before\_index:** Here, preparations for the **index** stage can be made, and artifacts from earlier stages can be cleaned up. For some languages, there are steps in between **after\_prepare**, and **before\_index**, which are discussed below. This stage does nothing by default.
4. **index:** In this stage, the extractor finally runs, and turns the identified relevant files into their LGTM representation called trap files (`.trap` file extension), which are included in the database together with copies of the source files they were generated from.

You may wonder why **after\_prepare** and **before\_index** are two separate stages, when they fulfill mostly the same purpose, and run right after each other. The reason for this is that they do not actually necessarily run consecutively, because there can be additional stages in between **after\_prepare** and **before\_index**, depending on the language being imported. If you are using a language without such an additional stage, it does not matter which of these stages you use, including using both.

- For C/C++ codebases, there is an additional stage called **configure** in between the **after\_prepare** and **before\_index** stages, which is used to generate configuration files that are needed for building the project in the **index** stage [10]. The default configuration tries to identify the build system by looking at the files present in the repository. A number of build systems are supported by default, but if this stage does not work out of the box, it can be manually configured.
- Python codebases have an additional extraction stage called **python\_setup**, which also sits in between the **after\_prepare** and **before\_index** stages. It is used to set up the Python interpreter and environment. You can choose between Python versions, and specify requirements to be installed by pip [53].

Figure 5 shows a flow diagram of the extraction pipeline with the stages described above.

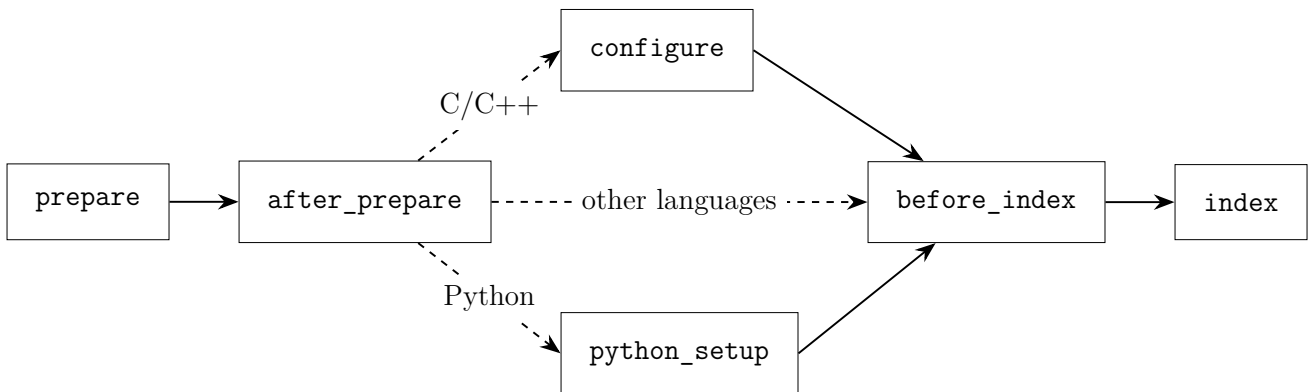


Figure 5: LGTM extraction pipeline

### 7.1.1 Importing Godot

This section demonstrates how to customize the LGTM configuration in order to import projects, with the Godot game engine [27] acting as an example.

The default C/C++ extraction configuration mostly works well for Godot. LGTM installs the requirements, and even picks up on the fact that Godot should be build using the SCons [60] build system. However, from consulting the import logs, we can see that the extraction fails at the `index` stage, because in order to build Godot, a target platform has to be specified [35].

By specifying a `build_command` for the `index` stage in the C/C++ extraction block, we can override the default configuration, and get LGTM to successfully build Godot [38]. The configuration file must be called `lgtm.yml` or `.lgtm.yml`, and it has to be included in the source code repository.

Listing 11: `lgtm.yml` configuration file for Godot

```
1 extraction :
2   cpp :
3     index :
4       build_command :
5         - scons platform=x11
```

After adding the configuration file (listing 11) to the Godot repository, it can be imported into LGTM without problems. As of 2019-03-23 this configuration has kindly been added to LGTM by Semmle’s Head of Product Bas van Schaik [61], and the project can now be queried on the LGTM website<sup>1</sup>.

As you can see, the import process can be very time consuming, because it requires understanding the build process of the projects. Even though LGTM offers default configurations for the supported languages, often customized configuration is required. Luckily, we have the sizable library of already imported projects at our disposal for the analysis.

## 7.2 Detecting Unsuitable Noise Sources

Our goal is detecting the use of unsuitable noise sources, which means we must identify functions that are used to seed a PRNG (henceforth referred to as *seed functions*), and are being called with data that is predictable.

Labeling all the low entropy sources manually is possible, but this approach would be very time consuming, and runs contrary to the goal of automating the vulnerability detection. Thus, we have to make use of heuristics. With all heuristics, there is a trade-off between sensitivity ( $\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$ ) and specificity ( $\frac{\text{true negatives}}{\text{true negatives} + \text{false positives}}$ ).

For the detection of unsuitable noise sources, prioritizing specificity over sensitivity makes sense, because

1. the bar for improvement is relatively low, so even detecting some vulnerabilities is already an improvement over the status quo, and

---

<sup>1</sup><https://lgtm.com/projects/g/godotengine/godot/overview/>

2. having low specificity negates the advantages of automation, because sifting through a large number of false positives is no substantial improvement over manually reviewing the code.

A good starting point for our analysis is identifying the seed functions, which are defined in the `SeedFunction` QL class (listing 12). After some experimentation with a heuristic for identifying seed functions, it was decided to just use a list of known seed functions from popular PRNG libraries, because while slightly increasing the sensitivity, using a heuristic also drastically decreased the specificity, which we decided to prioritize. The supported PRNG libraries are: GNU libc [65] (including ISO random, BSD random, and SVID random), C++ stdlib [11], OpenSSL [46], and PCG [48].

Listing 12: QL class: Seed Functions

```
1 class SeedFunction extends Function {
2   SeedFunction() {
3     // ISO random: https://www.gnu.org/software/libc/manual/html_node/
4     // ISO-Random.html
5     this.hasName("rand_r")
6     or this.hasName("srand")
7     // BSD random: https://www.gnu.org/software/libc/manual/html_node/
8     // BSD-Random.html
9     or this.hasName("initstate")
10    or this.hasName("initstate_r")
11    or this.hasName("srandom")
12    or this.hasName("srandom_r")
13    // SVID random: https://www.gnu.org/software/libc/manual/html_node/
14    // SVID-Random.html
15    or this.getName().matches("_rand48")
16    or this.getName().matches("_rand48\\_r")
17    or this.hasName("lcong48")
18    or this.hasName("lcong48_r")
19    or this.hasName("seed48")
20    or this.hasName("seed48_r")
21    // C++ stdlib
22    or this.hasName("seed")
23    // OpenSSL: https://www.openssl.org/docs/man1.1.1/man3/RAND_seed.
24    // html
25    or this.hasName("RAND_add")
26    or this.hasName("RAND_seed")
27    // PCG C library: http://www.pcg-random.org/using-pcg-c.html
28    or this.getName().matches("pcg%\\_srandom")
29    or this.getName().matches("pcg%\\_srandom\\_r")
30    // PCG C++ library: http://www.pcg-random.org/using-pcg-cpp.html
31    or this.getQualifiedName().matches("pcg%::pcg%")
32    or this.getQualifiedName().matches("pcg%::seed")
33  }
34 }
```

We want to scrutinize how these functions are being called, so we need to look at seed function calls. It follows from listing 12, that seed function calls are function calls with a seed function as their target, which is expressed in the QL class `SeedFunctionCall` (listing 13).

Listing 13: QL class: Seed Function Calls

```

1 class SeedFunctionCall extends FunctionCall {
2   SeedFunctionCall() { this.getTarget() instanceof SeedFunction }
3 }

```

The `SeedFunctionCall` class is already sufficient for some cursory analysis, like the following query (listing 14), which finds seed functions being called with a literal argument:

Listing 14: QL query: Seed Function Calls with Literal Arguments

```

1 import cpp
2
3 from SeedFunctionCall func_call
4 where func_call.getAnArgument() instanceof Literal
5 select func_call

```

Running this query (listing 14) yields a lot of false positives, because not all arguments of seed functions correspond to a required entropy input. Some arguments may just contain the length of other arguments, a personalization string, or some other auxiliary information. The specificity, which we decided to prioritize, is not high enough, resulting in many false positives being picked up. To rectify the query, we introduce the concept of *relevant parameters* (listing 15), which were taken from the documentation of the popular PRNG libraries we used [51, 52, 69, 47].

Listing 15: QL class: Relevant Parameters

```

1 class RelevantParameter extends Parameter {
2   RelevantParameter() {
3     // ISO random
4     this.getFunction().hasName("rand_r") // single parameter
5     or this.getFunction().hasName("srand") // single parameter
6     // BSD random
7     or (
8       this.getIndex() = 0 // all relevant parameters have index 0
9       and (
10        this.getFunction().hasName("initstate")
11        or this.getFunction().hasName("initstate_r")
12        or this.getFunction().hasName("srandom")
13        or this.getFunction().hasName("srandom_r")
14      )
15    )
16    // SVID random
17    or (
18      this.getIndex() = 0 // all relevant parameters have index 0

```

```

19     and (
20         this.getFunction().hasName("lcong48")
21         or this.getFunction().hasName("lcong48_r")
22         or this.getFunction().hasName("seed48")
23         or this.getFunction().hasName("seed48_r")
24         or this.getFunction().getName().matches("_rand48")
25         or this.getFunction().getName().matches("_rand48\\_r")
26     )
27 )
28 // C++ stdlib
29 or this.getFunction().hasName("seed") // single parameter
30 // OpenSSL
31 or this.getFunction().hasName("RAND_add") and this.getIndex() = 0
32 or this.getFunction().hasName("RAND_seed") and this.getIndex() = 0
33 // PCG C library
34 or this.getFunction().getName().matches("pcg%\\_srandom")
35     and this.getIndex() = 0
36 or this.getFunction().getName().matches("pcg%\\_srandom\\_r")
37     and this.getIndex() = 1
38 // PCG C++ library
39 or this.getFunction().getQualifiedName().matches("pcg%::pcg%")
40     and this.hasName("seed")
41 or this.getFunction().getQualifiedName().matches("pcg%::pcg%")
42     and this.hasName("seq")
43 or this.getFunction().getQualifiedName().matches("pcg%::seed")
44     and this.hasName("seed")
45 or this.getFunction().getQualifiedName().matches("pcg%::seed")
46     and this.hasName("seq")
47 }
48 }

```

It follows that relevant arguments are arguments from a seed function call that correspond to a relevant parameter of the target function, which is codified in listing 16.

Listing 16: QL class: Relevant Arguments

```

1 class RelevantArgument extends Expr {
2     RelevantArgument() {
3         exists(SeedFunction sf, SeedFunctionCall sf_call, int i |
4             sf_call.getTarget() = sf
5             and sf_call.getArgument(i) = this
6             and sf.getParameter(i) instanceof RelevantParameter
7         )
8     }
9 }

```

Many other false positives are caused by the fact that predetermined seeds are often consciously chosen in testing code, in order to facilitate reproducibility, so these cases should be filtered out as

well. That is handled by the `IgnoredFile` class (listing 17), which contains a heuristic rejecting probably irrelevant files.

Listing 17: QL class: Ignored Files

```
1 class IgnoredFile extends File {
2     IgnoredFile() {
3         this instanceof TestFile
4         or this.getBaseName().matches("%test%")
5         or this.getBaseName().matches("%bench%")
6     }
7 }
```

Making use of these ingredients, we can already put together quite a sophisticated query (listing 18). Instead of looking only for literals, we can also widen our search to include all constants, now that we have achieved a far higher specificity.

Listing 18: QL query: Seed Function Calls with constant Relevant Arguments

```
1 import cpp
2 import semmle.code.cpp.TestFile
3
4 from SeedFunctionCall sf_call, RelevantArgument arg
5 where arg.isConstant()
6     and sf_call.getAnArgument() = arg
7     and not sf_call.getFile() instanceof IgnoredFile
8 select sf_call, arg
```

From the examples (section 4), we know that often low entropy arguments are not constants, but functions returning low entropy outputs, like in the common anti-pattern `srand(time(NULL))`. To catch cases like this, let us introduce the concept of *low entropy functions* (listing 19), and *low entropy function calls* (listing 20).

Listing 19: QL class: Low Entropy Functions

```
1 class LowEntropyFunction extends Function {
2     LowEntropyFunction() {
3         this.hasName("now")
4         or this.hasName("pid")
5         or this.hasName("ppid")
6         or this.getName().matches("%time%")
7         or this.getName().matches("%ticks%")
8     }
9 }
```

For detecting low entropy functions, a whitelist is used in combination with a heuristic based on the function name containing some keywords like “time”. The LGTM staff was kind enough to offer an assortment of 69 sample projects to test the queries on, and this combination appears to



work really well in practice, picking up a lot of low entropy functions, without any false positives in the test set. The full set of sample project is included in Appendix C: LGTM Sample Projects.

Listing 20: QL class: Low Entropy Function Calls

```

1 class LowEntropyFunctionCall extends FunctionCall {
2     LowEntropyFunctionCall() {
3         this.getTarget() instanceof LowEntropyFunction
4     }
5 }

```

Using the `LowEntropyFunctionCall` class, we can now also find problems arising from PRNGs being seeded with low entropy functions, as shown in listing 21:

Listing 21: QL query: Seed Function Calls with Low Entropy Function as Relevant Argument

```

1 import cpp
2 import semmle.code.cpp.TestFile
3
4 from SeedFunctionCall sf_call, RelevantArgument arg
5 where arg instanceof LowEntropyFunctionCall
6     and sf_call.getAnArgument() = arg
7     and not sf_call.getFile() instanceof IgnoredFile
8 select sf_call, arg

```

In pursuit of covering more complicated cases like `seed(time(NULL) ^ 1337)`, in which constants and low entropy function calls are combined in arbitrary operations, we introduce the mutually recursive classes `LowEntropySource` (listing 22), and `LowEntropyOperation` (listing 23).

Listing 22: QL class: Low Entropy Source

```

1 class LowEntropySource extends Expr {
2     LowEntropySource() {
3         this.isConstant()
4         or this instanceof LowEntropyFunctionCall
5         or this instanceof LowEntropyOperation
6     }
7 }

```

A *low entropy source* is defined as being either a constant, a low entropy function call, or a *low entropy operation*, which itself is an operation with only low entropy sources as operands (figure 6).

Listing 23: QL class: Low Entropy Operation

```

1 class LowEntropyOperation extends Expr {
2   LowEntropyOperation() {
3     this instanceof Operation and
4     forall(Expr operand | this.(Operation).getAnOperand() = operand |
5       operand instanceof LowEntropySource
6   )
7 }
8 }

```

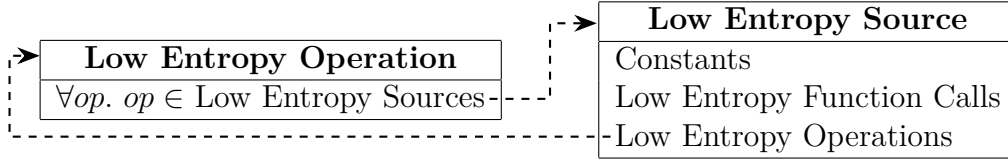


Figure 6: Mutually recursive relation between low entropy sources and low entropy operations

With this framework in place, we can now formulate a query which finds a wide range of low entropy sources being used as arguments to seed functions, as shown in listing 24.

Listing 24: QL query: Seed Function Calls with Low Entropy Source as Relevant Argument

```

1 import cpp
2 import semmle.code.cpp.TestFile
3 import semmle.code.cpp.dataflow.DataFlow
4
5 from SeedFunctionCall sf_call, RelevantArgument argument
6 where not sf_call.getFile() instanceof IgnoredFile
7     and sf_call.getAnArgument() = argument
8     and argument instanceof LowEntropySource
9 select sf_call, argument

```

The queries in this section (listings 14, 18, 21, and 24) already match a number of security vulnerabilities with impressive specificity, but we can do much better in terms of sensitivity, as demonstrated in section 7.3.

### 7.3 Data Flow Analysis

The queries shown in section 7.2 are already a good basis for our analysis, but are very limited in their scope. They are able to identify cases in which seed functions are being called directly with constants (e.g.  `srand(23)`), low entropy functions (e.g.  `srand(time(NULL))`), or low entropy operations (e.g.  `srand(time(NULL) ^ 42)`) as their arguments. In reality, most vulnerabilities are not that obvious, and they look more like this:

Listing 25: More typical vulnerability

```

1 int seed;
2 if (some_condition) {
3     seed = secure_noise_source();
4 } else if (another_condition) {
5     seed = 322;
6 } else {
7     seed = time(NULL);
8 }
9 srand(seed);

```

Keeping track of all the possible values a variable could hold with the tools we have used so far would theoretically be possible though very cumbersome. In real code, vulnerabilities are often much more complicated than this, and can involve multiple file and function call boundaries. Fortunately, LGTM offers excellent *data flow analysis* libraries [34], which aid us in writing exactly these sorts of queries.

“Data flow analysis computes the possible values that a variable can hold at various points in a program, determining how those values propagate through the program and where they are used.” [34]

LGTM represents this information as *data flow graphs*, in which data flows from *sources* to *sinks*. Computing these graphs as a prerequisite for data flow analysis presents several challenges, and can get highly resource intensive. Certain information is only available at runtime, and requires the data flow libraries to perform additional work in order to determine call targets. Aliasing complicates the analysis further, because a single memory location can be aliased to several pointers, and writes to such a memory location affect all of them. For these reasons, LGTM offers two different flavors of data flow analysis.

*Local data flow* concerns data flow within a single function, and is much cheaper to analyze and more accurate compared to *global data flow*, which deals with data flow across an entire program. For many queries, local data flow analysis is sufficient, but some queries necessitate the use of global data flow analysis. In that case, sources and sinks of data flows should be prefiltered in order to make the performance more palatable.

Before we can use the data flow library, we need to import the C/C++ specific version of the library `semmle.code.cpp.dataflow.DataFlow` [6]. Each language comes with its own data flow library, for the same reason they come with their own extractor. The languages are too diverse for a unified representation to do justice to all the language features. Having separate libraries guarantees maximum accuracy.

Next we can prepare the data flow analysis by telling LGTM which sources and sinks we are interested in, by extending the `DatFlow::Configuration` class representing data flow configurations, and overriding the `isSource` and `isSink` predicates (listing 26). This configuration step is necessary because we want to use global data flow analysis, which is more conducive to high sensitivity compared to local data flow analysis.

Listing 26: QL class: Low Entropy Flows

```

1 class LowEntropyFlow extends DataFlow::Configuration {
2   LowEntropyFlow() { this = "LowEntropyFlow" }
3   override predicate isSource(DataFlow::Node source) {
4     source.asExpr() instanceof LowEntropySource
5   }
6   override predicate isSink(DataFlow::Node sink) {
7     sink.asExpr() instanceof RelevantArgument
8   }
9 }

```

We want to investigate data flows from low entropy sources to relevant arguments, so that is what we filter by. With the configuration out of the way, we can finally implement our last query, checking for data flow from low entropy sources to relevant arguments (listing 27).

Listing 27: QL query: Data Flow from Low Entropy Source to Relevant Argument

```

1 import cpp
2 import semmle.code.cpp.TestFile
3 import semmle.code.cpp.dataflow.DataFlow
4
5 from LowEntropyFlow flow_conf, SeedFunctionCall sf_call,
   RelevantArgument sink, LowEntropySource source
6 where not sf_call.getFile() instanceof IgnoredFile
7   and not source.getFile() instanceof IgnoredFile
8   and sf_call.getAnArgument() = sink
9   and flow_conf.hasFlow(
10     DataFlow::exprNode(source), DataFlow::exprNode(sink)
11   )
12 select sf_call, source

```

The query we have developed in this section is able to robustly identify a wide range of vulnerabilities, by globally tracing data flow from diverse low entropy sources to relevant arguments of seed functions from many popular PRNG libraries. When run against the 69 sample projects (section 9.3), it finds 74 separate instances of unsuitable noise sources used as relevant arguments, in 20 of the projects.

For some of these instances, it is difficult to say whether the behavior is intentional, which could be classified as a bug regardless of whether it leads to a vulnerability. If a bad noise source is used, it should be immediately obvious that this is safe and intentional (like in testing code). For example, the Bullet physics engine [9], which is included as a dependency in many projects, contains this call: `seed(1806)` in the `btSoftBodyHelpers.cpp` file. There are no comments explaining why a constant seed is chosen, and the call is made in non-testing code, making this either a vulnerability, or a documentation bug. Overall, no obvious false positives are found in the test set.

Once the query had been finalized on the test set of 69 sample projects, Bas van Schaik [61] kindly offered to perform an analysis on the whole LGTM project catalogue, containing about

8000 C/C++ projects. Unfortunately, this is not possible for normal LGTM users, due to the high resource cost of performing such a comprehensive analysis, but the LGTM team is extremely approachable.

## 7.4 Analysis Results

Out of the about 8000 C/C++ projects the analysis was run on, 1229 (~15%) used at least one bad noise source to seed a PRNG, and are henceforth referred to as *vulnerable projects*. Regardless of whether these PRNG seeding failures directly lead to exploitable (security) vulnerabilities, this classification is justified, because even if badly seeded PRNGs do not directly lead to exploitable weaknesses, they unnecessarily expose attack surfaces in projects they are used in (see section 4.3), requiring increased diligence from the programmers, which is impossible to keep up indefinitely. If there are no badly seeded PRNGs in a project, they cannot be misused, and the responsibility of generating secure seeds can and should be deferred to the host operating system in almost all cases (section 5.5).

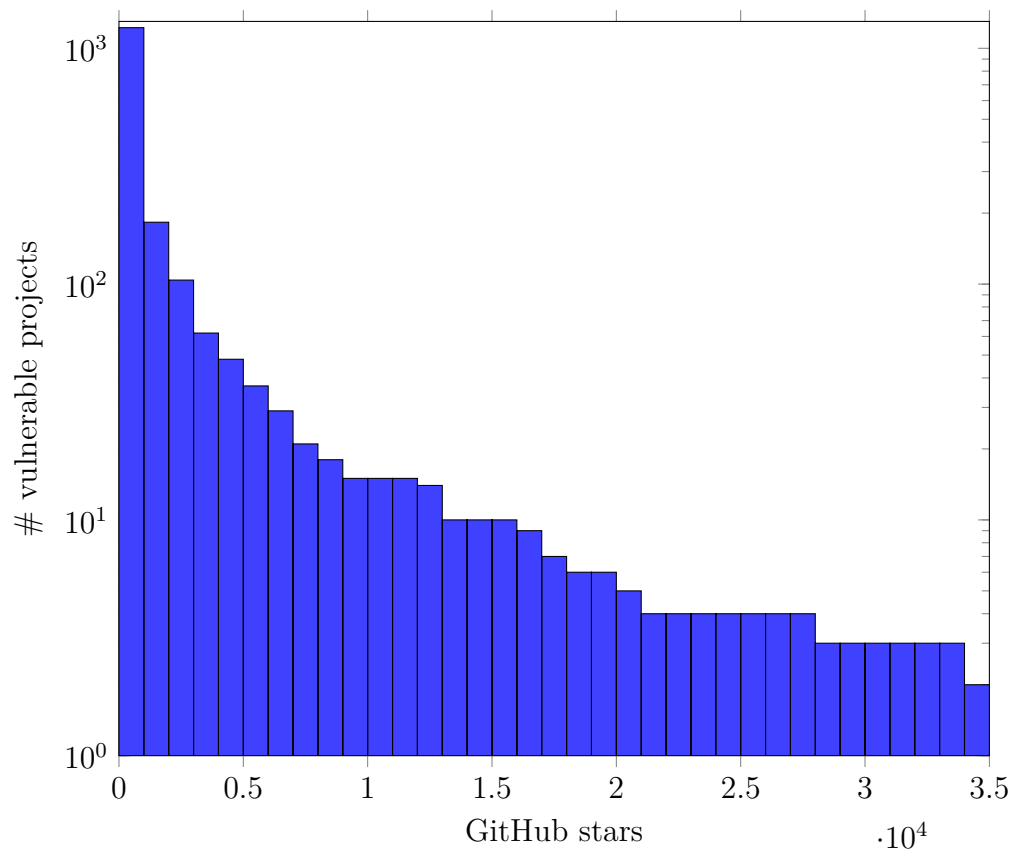


Figure 7: Popularity of the projects containing bad noise sources

From the 1229 vulnerable projects, 1224 were hosted on GitHub, 1223 of which are still available. The `TheZ3ro/c-3po` repository appears to have been deleted, so the number of stars it had could not be retrieved. That means we know the number of GitHub stars for over 99.5% of vulnerable projects. Figure 7 plots GitHub stars against the number of vulnerable projects with at least that many stars (in steps of 1000 stars), and shows that the vast majority of bad noise sources were found in projects having a low number of stars on GitHub, which can be seen as a rough indicator

of their popularity. It is likely that most of these unpopular projects with a low number of stars are never going to be manually audited, emphasizing the importance of automation. The number of popular projects on GitHub is low in general, so it makes sense that the number of popular vulnerable projects is low as well. However, included in the vulnerable projects are 183 projects with more than 1000 stars, and even 15 projects with more than 10 000 stars, such as the very popular projects Redis, scikit-learn, OpenCV, Godot, and vim.

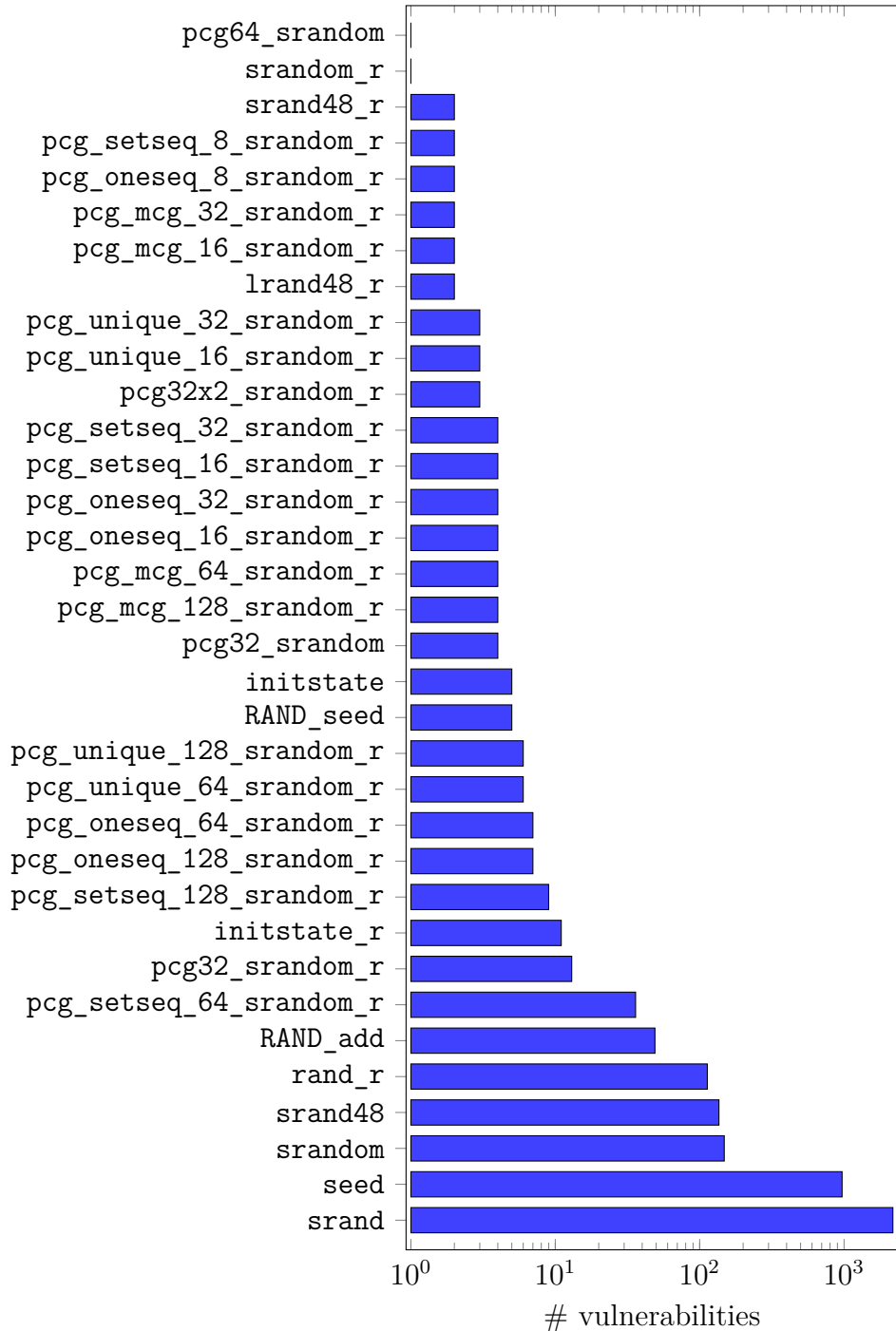


Figure 8: Seed Function Frequencies

34 different seed functions with low entropy data flow to one of their relevant arguments have been

detected in all of the analyzed projects. As visualized in figure 8, the overwhelming majority of the seed function calls with low entropy arguments had the `srand` function from the C standard library, or the `seed` function from the C++ standard library as their targets. Occurrence frequencies drop off sharply thereafter, with a long tail of seldom used seed functions, many of which coming from the PCG family [43].

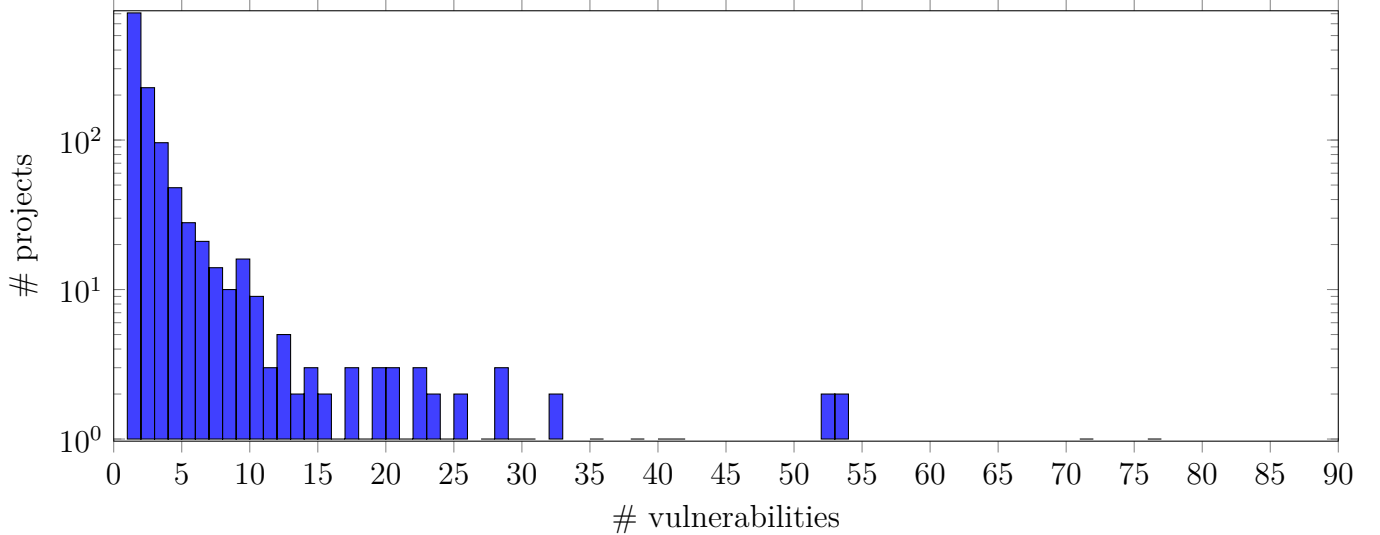


Figure 9: Number of vulnerabilities per project

The majority of vulnerable projects, consisting of 709 individuals, contained only a single vulnerability, with a long tail of projects containing up to 90 bad seed function calls in one case (figure 9). Fixing most of the affected projects would be as simple as replacing a single insecure noise source with a better one, like the one provided by the host operating system (section 5.5). This also underlines the importance of automating the vulnerability detection (as accomplished in sections 7.2, and 7.3), as manually reviewing hundreds of projects containing a single vulnerability each is impractical, but with variant analysis this task becomes trivial.

Figure 10 illustrates the low entropy source occurrence frequencies. The full list of low entropy sources from the analysis is too long to be reproduced in full here, with a large number of rarely used sources, so the figure contains the 20 most commonly used ones. The most frequent low entropy source by an order of magnitude (with 1711 occurrences) are calls to `time()`, which is a common anti-pattern we have already identified earlier (section 6.1). After that we have mostly constants like 0 or 1, and low entropy operations like the negation operation `- ...`, or the bitwise exclusive or operation `... ^ ...`. The `LowEntropySource` class (listing 22) seems to be working well, and is picking up a varied assortment of bad entropy sources.

For the remainder of this section, we will focus on the most popular vulnerable projects in greater detail.

## Redis

With over 35 000 stars on GitHub, Redis [54], an open source in-memory database, is the most popular vulnerable project which was identified during the analysis. It is relied on by many

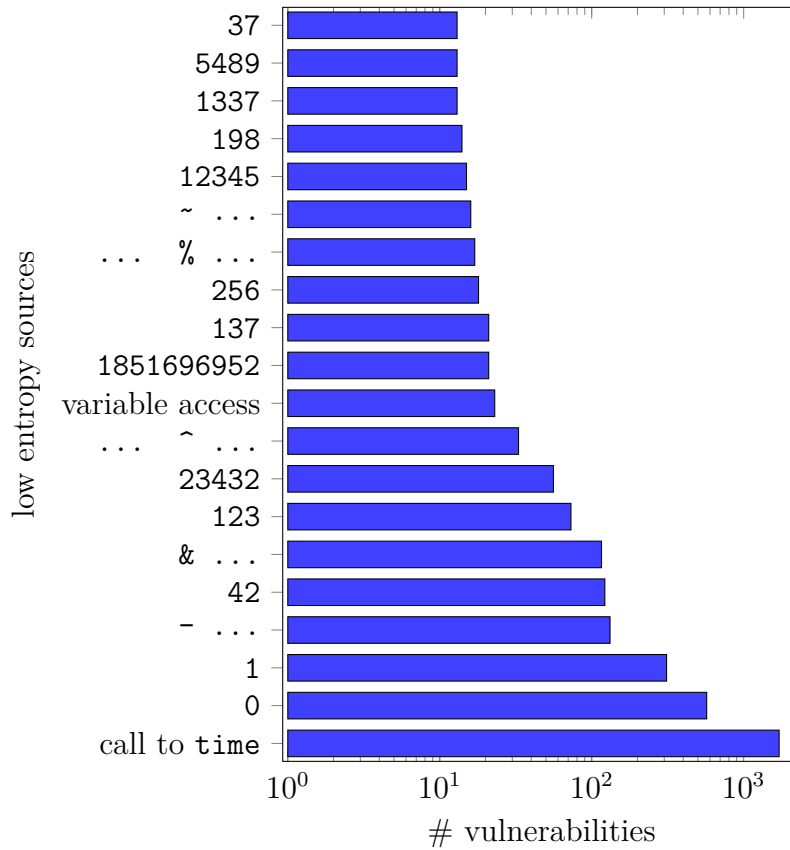


Figure 10: Low Entropy Source Frequencies

high profile corporations, including *Microsoft*, *American Express*, and *Master Card* [55], so an exploitable vulnerability could lead to critical infrastructure outages.

During the analysis, 4 instances of improper seeding of random number generators were found, all concerning the `srand` C standard library ISO seeding function. Three of these are in the Redis code itself, and another one in Lua, which is included as a dependency. Let us get the dependency out of the way, so that we can focus on Redis’ own code. The query identified the function call to `srand` shown in listing 28 as potentially being called with the literal 0.

Listing 28: Redis vulnerability #1 part 1 (excerpt from `deps/lua/src/lmathlib.c`) [56]

```

208 // ...
209 static int math_randomseed (lua_State *L) {
210     srand(luaL_checkint(L, 1));
211     return 0;
212 }
213 // ...

```

If we look at the definition of the `luaL_checkinteger` function (listing 29), we see that it uses `luaL_tointeger`, which can return 0 (listing 30). `luaL_checkinteger` does check for the return value being 0, but then passes it on anyway, leading to the vulnerability.



Listing 29: Redis vulnerability #1 part 2 (excerpt from `deps/lua/src/lauxlib.c`) [56]

```

188 // ...
189 LUALIB_API lua_Integer luaL_checkinteger (lua_State *L, int nargs) {
190     lua_Integer d = lua_tointeger(L, nargs);
191     if (d == 0 && !lua_isnumber(L, nargs)) /* avoid extra test when d is
        not 0 */
192         tag_error(L, nargs, LUA_TNUMBER);
193     return d;
194 }
195 // ...

```

Listing 30: Redis vulnerability #1 part 3 (excerpt from `deps/lua/src/lapi.c`) [56]

```

322 // ...
323 LUALIB_API lua_Integer lua_tointeger (lua_State *L, int idx) {
324     TValue n;
325     const TValue *o = index2adr(L, idx);
326     if (tonumber(o, &n)) {
327         lua_Integer res;
328         lua_Number num = nvalue(o);
329         lua_number2integer(res, num);
330         return res;
331     }
332     else
333         return 0;
334 }
335 // ...

```

The three remaining vulnerabilities in Redis were all found in the `src/redis-cli.c` file, which is responsible for the command line interface. They are all manifestations of the same vulnerable seeding method `srand(time(NULL))`, confirming the abundance of this anti-pattern, which we have identified and discussed in section 6.1.

## scikit-learn

scikit-learn [59] is a machine learning framework for Python, based on NumPy, SciPy, and matplotlib. With over 34 thousand GitHub stars, it is the second most popular vulnerable project by this metric, and is used by companies like *JPMorgan Chase and Co.*, and *Spotify* [71].

22 vulnerabilities have been detected, all referencing a single function call to `srand` in the `sklearn/svm/src/liblinear/liblinear_helper.c` file, specifically within the `set_parameter()` helper function (listing 31).

Listing 31: scikit-learn vulnerability (excerpt from `liblinear_helper.c`) [58]

```

184 // ...
185 /* Create a paramater struct with and return it */
186 struct parameter *set_parameter(
187     int solver_type, double eps, double C, npy_intp nr_weight,
188     char *weight_label, char *weight, int max_iter, unsigned seed,
189     double epsilon
190 ) {
191     struct parameter *param = malloc(sizeof(struct parameter));
192     if (param == NULL)
193         return NULL;
194
195     srand(seed);
196     param->solver_type = solver_type;
197     param->eps = eps;
198     param->C = C;
199     param->p = epsilon; // epsilon for epsilon-SVR
200     param->nr_weight = (int) nr_weight;
201     param->weight_label = (int *) weight_label;
202     param->weight = (double *) weight;
203     param->max_iter = max_iter;
204     return param;
205 }
206 // ...

```

The source for all 22 vulnerabilities lies within the over 9500 lines long auto-generated file `sklearn/svm/liblinear.c`, which does indeed contain a call to `set_parameter()`, but is otherwise utterly incomprehensible. `sklearn/svm/liblinear.c` is generated during the build process, in such a way that it is not really humanly readable, and thus not reasonably manually auditable. It does, however, contain a number of other oddities, like empty `else` clauses. These vulnerabilities were not investigated much further, because in any case there is definitely a problem here. It is pretty improbable – if not impossible – that anyone understands what this file does in its entirety, and it is therefore almost guaranteed to contain bugs of some sort. The LGTM data flow libraries can probably be trusted when they say that the file contains 22 improper seed function calls.

## OpenCV

The third most popular vulnerable project, according to its number of GitHub stars, is the open source computer vision library OpenCV [44]. With over 33 000 stars, it is not far behind Redis and scikit-learn, which is not surprising. With over 2500 optimized implementations of algorithms ranging from classical to state of the art, and a permissive license, it has become a top choice when it comes to computer vision libraries.

The analysis flagged up a single vulnerable call to `srand` in the `modules/core/src/rand.cpp` file, which contains a constructor for a Mersenne Twister PRNG with a constant seed (5489) (listing 32). This should definitely be improved with a proper noise source, because unsuspecting users might reasonably expect the PRNG to be properly seeded when using this constructor. When

seeded with a constant, the Mersenne Twister algorithm produces predictable output, which is generally not desired from a PRNG.

Listing 32: OpenCV vulnerability (excerpt from `rand.cpp`) [45]

```
817 // ...
818 cv::RNG_MT19937::RNG_MT19937() { seed(5489U); }
819 // ...
```

Investigating the matches for several high-profile projects in detail has proved that the QL query, which we developed in sections 7.2 and 7.3, reliably captures real vulnerabilities, some of which are extremely hard to find manually.

## 8 Conclusions

Addressing the common need of generating randomness, increasingly sophisticated PRNG algorithms have been developed over the last few years, offering excellent statistical performance and cryptographic security, with a low memory and time complexity. There is no excuse for not using a secure random number generator in this day and age, even on embedded and IoT devices with extremely limited resources. Securely seeding these generators is still a problem, and their security model relies on the seed being kept secret and hard to predict.

Weaknesses in entropy generation remain widespread (section 7.4), even though best practices for entropy generation exist, and have been implemented in all mainstream operating systems, all of which expose interfaces to cryptographically secure and well tested PRNGs, offering a solid base for seeding PRNGs. Nonetheless, many popular projects still use bad entropy generation schemes, which unnecessarily exposes attack surfaces, and the potential for security vulnerabilities. Large scale efforts to improve the situation over several years have largely failed [32, 30], making it clear that a different approach is needed to effectively improve the state of entropy generation in practice.

In section 7, such a novel approach is presented, which combines variant analysis (section 7.2) with data flow analysis (section 7.3), in order to identify low entropy sources being used as relevant seed function arguments. This takes the form of a QL query, which can be run on the LGTM code analysis platform. Analyzing all of the about 8000 C/C++ projects which have been imported into LGTM with this method reveals that the developed QL query is effective in detecting even relatively complicated cases of data flowing from unsuitable noise sources to relevant arguments of seed functions, which can be an invaluable tool in pushing back against the prevalence of insecure entropy generation (section 7.4).

In spite of the query already being quite effective, there is still room for improvement; for example by detecting even more forms of data flow, by supporting other programming languages, or by broadening the variant analysis to include other cases of entropy generation best practices being disregarded. I was declared *New User of the Month* for April 2019 by LGTM [50], whose staff is still collaborating with me on improving the query, which is under consideration for inclusion in the LGTM standard query catalogue. Once it has been accepted, everybody can profit from the insights gained in this thesis (in addition to the whole array of queries which LGTM already offers) without much effort, by simply importing their project into LGTM. This represents one of the main advantages of automating vulnerability detection with tools like LGTM.

It remains to be seen whether variant analysis and data flow analysis will effectively raise the standard of entropy generation in practice, considering that previous efforts have largely failed in doing so; though they definitely have the potential of sustainably increasing the security of entropy generation, and thus random number generation. The ineffectiveness of earlier approaches to improve the state of affairs with regards to entropy generation can largely be attributed to them failing to be accepted by the community, and incorporated into standard quality assurance practices. Lowering the bar of entry for applying the techniques developed in this thesis, by offering them in an easy to use package like LGTM, should benefit their uptake, and hopefully lead to increased efficacy.

## 9 Appendix

### 9.1 A: Proof of Concept

This section contains auxiliary files for the proof of concept presented in section 5.1.

Listing 33: util.h

```
1 #define NSAMPLES 10
2
3 time_t modification_time(char const *filename);
4 FILE *open_file(char const *filename, char const *mode);
```

Listing 34: util.c

```
1 #include <errno.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #include <sys/stat.h>
6
7 /**
8  * Retrieve the time of last modification of some file, and check for
9  * errors.
10  *
11  * @param filename path to the file
12  *
13  * @return time of last modification
14  */
15 time_t modification_time(char const *filename) {
16     struct stat stats;
17     if (stat(filename, &stats)) {
18         fprintf(stderr, "Could not retrieve information about file \"%s\":\n",
19                 filename, errno);
20         exit(EXIT_FAILURE);
21     }
22     return stats.st_mtim.tv_sec;
23 }
24
25 /**
26  * Open a file, and check for errors.
27  *
28  * @param filename path to the file
29  * @param mode file access mode
30  *
31  * @return file pointer
32  */
33 FILE *open_file(char const *filename, char const *mode) {
34     FILE *file = fopen(filename, mode);
```

```

34     if (!file) {
35         fprintf(stderr, "Could not open file \"%s\": %d\n", filename,
36             errno);
37         exit(EXIT_FAILURE);
38     }
39     return file;

```

Listing 35: Makefile

```

1 CC ?= gcc
2 CFLAGS += -Wall -Wextra
3
4 all: attack poc
5
6 %: %.c util.c
7     $(CC) $(CFLAGS) -o $@ $^

```

## 9.2 B: Emails from Christian Uldall Pedersen

Some of the information in the section about Godot (section 4.4) was taken from emails by Christian Uldall Pedersen, who is the CTO of Gamblify.

Listing 36: Email from 2019-03-12

```

1 FROM: cup@gamblify.com
2 TO: hendrik.to@gmail.com
3
4 Hi Hendrik,
5
6 Thank you for your email, and interest in our games.
7
8 Gamblify actually only use Godot for the graphical part of our slot-
9 machine games. The actual randomness is done using a combination of
10 C++ and Java, where we use C++ for the seed generation, and Java for
11 generating pseudo randomness based on this seed.
12
13 In C++ we use C++11's std::random_device, while in Java we use "new
14 Random(long)": https://docs.oracle.com/javase/8/docs/api/java/util/
    Random.html#Random-long-
15
16 The reason why we use a pseudo random number generator is to be able
17 to playback our game logic given a specific seed.
18
19 I would love to hear from you, if you have any comments on this
20 approach.

```

```
21
22 Best regards ,
23 Christian Uldall Pedersen
24 CTO
25 Gamblify
```

Listing 37: Email from 2019-03-13

```
1 FROM: cup@gamblify.com
2 TO: hendrik.to@gmail.com
3
4 Hi Hendrik ,
5
6 We are only sending a description of what result to show in the Godot
7 frontend. No randomness is sent. We implemented a TCP based integra-
8 tion for this.
9
10 You are welcome to quote me. I would like to have it sent for
11 approval.
12
13 Good luck with your thesis!
14
15 Best regards ,
16 Christian
```

Listing 38: Email from 2019-03-18

```
1 FROM: cup@gamblify.com
2 TO: hendrik.to@gmail.com
3
4 Hi Hendrik ,
5
6 We are using java.util.Random constructed with the random seed gener-
7 ated from the C++ code. The purpose is that we want our Java "engine"
8 to behave as a function. I.e. same input yields same output. This re-
9 quires a deterministic PRNG. The "true" random seed from the C++ code
10 ensures that the outcomes on the machine are random. Giving the Java
11 engine this property makes it much easier to debug potential errors ,
12 i.e. we can replay an outcome in case of a dispute.
13
14 I approve the quotation :)
15
16 Best regards ,
17 Christian
```

I want to thank Christian for being very open, swiftly answering my questions, and for giving his permission for me to quote his answers for my thesis.

### 9.3 C: LGTM Sample Projects

The following is an exhaustive list of the 69 sample projects generously offered by the LGTM staff for testing the QL queries.

- [assimp/assimp](#)
- [autopilot-rs/autopy-legacy](#)
- [awslabs/s2n](#)
- [bingmann/cryptote](#)
- [Blizzard/s2client-api](#)
- [bloomberg/comdb2](#)
- [casadi/casadi](#)
- [chriscamacho/gles2framework](#)
- [coreutils/coreutils](#)
- [cossacklabs/themis](#)
- [couchbase/libcouchbase](#)
- [csete/gqrx](#)
- [curl/curl](#)
- [cxong/cdogs-sdl](#)
- [davidstutz/superpixels-revisited](#)
- [diffblue/cbmc](#)
- [dns-stats/compactor](#)
- [domoticz/domoticz](#)
- [dotnet/coreclr](#)
- [Enlightenment/efl](#)
- [fawkesrobotics/fawkes](#)
- [FFmpeg/FFmpeg](#)
- [fish-shell/fish-shell](#)
- [GNOME/libgnome-keyring](#)
- [godotengine/godot](#)
- [google/breadboard](#)
- [hishamhm/htop](#)
- [horde3d/Horde3D](#)
- [jbj/magicrescue](#)
- [jedisct1/libsodium](#)
- [KhronosGroup/VK-GL-CTS](#)
- [lastpass/lastpass-cli](#)
- [libretro/RetroArch](#)
- [LibtraceTeam/libtrace](#)
- [lightspark/lightspark](#)
- [logrotate/logrotate](#)
- [LuaJIT/LuaJIT](#)
- [lvmteam/lvm2](#)
- [madler/zlib](#)
- [ManaPlus/ManaPlus](#)
- [mbroz/cryptsetup](#)
- [Microsoft/ChakraCore](#)
- [MidnightCommander/mc](#)
- [miguelfreitas/twister-core](#)
- [musescore/MuseScore](#)
- [nishadg246/pybullet-play](#)
- [nlohmann/json](#)
- [ntpsec/ntpsec](#)
- [numpy/numpy](#)
- [pgsql-jp/jpug-doc](#)
- [pmacct/pmacct](#)
- [PowerDNS/pdns](#)



- [protocolbuffers/protobuf](#)
- [pwsafe/pwsafe](#)
- [qayshp/TestDisk](#)
- [sustrik/libmill](#)
- [systemd/systemd](#)
- [taglib/taglib](#)
- [tesseract-ocr/tesseract](#)
- [thegenemyers/DALIGNER](#)
- [thestk/stk](#)
- [torvalds/linux](#)
- [vim/vim](#)
- [VowpalWabbit/vowpal\\_wabbit](#)
- [wireshark/wireshark](#)
- [xiph/vorbis](#)
- [Z3Prover/z3](#)
- [zealdocs/zeal](#)
- [zeromq/libzmq](#)

## Figures

1	Rényi entropy measures . . . . .	6
2	NIST SP 800-90B Noise Source . . . . .	9
3	NIST SP 800-90B Entropy Source . . . . .	10
4	Overview of Entropy Generation Weaknesses . . . . .	28
5	LGTM extraction pipeline . . . . .	34
6	Mutually recursive relation between low entropy sources and low entropy operations	41
7	Popularity of the projects containing bad noise sources . . . . .	44
8	Seed Function Frequencies . . . . .	45
9	Number of vulnerabilities per project . . . . .	46
10	Low Entropy Source Frequencies . . . . .	47

# Listings

1	7-Zip 18.06 PRNG ( <code>RandGen.cpp</code> ) [2]	16
2	7-Zip 18.06 PRNG ( <code>Random.cpp</code> ) [2]	19
3	Godot RNG (excerpt from <code>random_pcg.cpp</code> ) [28]	19
4	Godot RNG (excerpt from <code>random_pcg.h</code> ) [28]	20
5	Godot RNG ( <code>pcg.cpp</code> ) [28]	20
6	Generator ( <code>poc.c</code> )	22
7	Attacker ( <code>attack.c</code> )	23
8	QL query: Pythagorean triples	32
9	QL query: Pythagorean triples with object-orientation	32
10	QL query: longest function name (C/C++)	33
11	<code>lgtm.yml</code> configuration file for Godot	35
12	QL class: Seed Functions	36
13	QL class: Seed Function Calls	37
14	QL query: Seed Function Calls with Literal Arguments	37
15	QL class: Relevant Parameters	37
16	QL class: Relevant Arguments	38
17	QL class: Ignored Files	39
18	QL query: Seed Function Calls with constant Relevant Arguments	39
19	QL class: Low Entropy Functions	39
20	QL class: Low Entropy Function Calls	40
21	QL query: Seed Function Calls with Low Entropy Function as Relevant Argument	40
22	QL class: Low Entropy Source	40
23	QL class: Low Entropy Operation	41
24	QL query: Seed Function Calls with Low Entropy Source as Relevant Argument	41
25	More typical vulnerability	42
26	QL class: Low Entropy Flows	43
27	QL query: Data Flow from Low Entropy Source to Relevant Argument	43
28	Redis vulnerability #1 part 1 (excerpt from <code>deps/lua/src/lmathlib.c</code> ) [56]	47
29	Redis vulnerability #1 part 2 (excerpt from <code>deps/lua/src/lauxlib.c</code> ) [56]	48
30	Redis vulnerability #1 part 3 (excerpt from <code>deps/lua/src/lapi.c</code> ) [56]	48
31	scikit-learn vulnerability (excerpt from <code>liblinear_helper.c</code> ) [58]	49
32	OpenCV vulnerability (excerpt from <code>rand.cpp</code> ) [45]	50
33	<code>util.h</code>	52
34	<code>util.c</code>	52
35	<code>Makefile</code>	53
36	Email from 2019-03-12	53
37	Email from 2019-03-13	54
38	Email from 2019-03-18	54

# Bibliography

- [1] *[RFC,-v2] random: introduce `getrandom(2)` system call*. Linux Kernel Mailing List. URL: <https://lore.kernel.org/patchwork/patch/484605/> (visited on 2019-03-17).
- [2] *7-Zip Source Code*. version 18.06. URL: <https://sourceforge.net/projects/sevenzzip/files/7-Zip/18.06/7z1806-src.7z/download> (visited on 2019-02-10).
- [3] *7-Zip Source Code*. version 19.00. URL: <https://sourceforge.net/projects/sevenzzip/files/7-Zip/19.00/7z1900-src.7z/download> (visited on 2019-03-23).
- [4] *About LGTM - Help - LGTM*. URL: <https://lgtm.com/help/lgtm/about-lgtm> (visited on 2019-03-17).
- [5] *About QL — Learn QL*. URL: <https://help.semmle.com/QL/learn-ql/ql/about-ql.html> (visited on 2019-03-17).
- [6] *Analyzing data flow in C/C++ - Help - LGTM*. URL: <https://lgtm.com/help/ql/cpp/dataflow> (visited on 2019-03-27).
- [7] Boaz Barak and Shai Halevi. “A Model and Architecture for Pseudo-random Generation with Applications to `/dev/random`”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. Alexandria, VA, USA: ACM, 2005, pp. 203–212. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102148. URL: <http://doi.acm.org/10.1145/1102120.1102148>.
- [8] H. Blasbalg and R. van Blerkom. “Message Compression”. In: *IRE Transactions on Space Electronics and Telemetry* SET-8 (Sept. 1962), pp. 228–238. DOI: 10.1109/IRET-SET.1962.5008841.
- [9] *Bullet Real-Time Physics Simulation / Home of Bullet and PyBullet: physics simulation for games, visual effects, robotics and reinforcement learning*. URL: <http://bulletphysics.org> (visited on 2019-03-29).
- [10] *C/C++ extraction - Help - LGTM*. URL: <https://lgtm.com/help/lgtm/cpp-extraction> (visited on 2019-03-21).
- [11] *C++ Standard Library headers - cppreference.com*. URL: <https://en.cppreference.com/w/cpp/header> (visited on 2019-03-28).
- [12] Alonzo Church. “On the concept of a random sequence”. In: *Bulletin of the American Mathematical Society* 46.2 (Feb. 1940), pp. 130–135. URL: <https://projecteuclid.org:443/euclid.bams/1183502434>.
- [13] Rudolf Clausius. “Ueber verschiedene für die Anwendung bequeme Formen der Hauptgleichungen der mechanischen Wärmetheorie”. In: *Annalen der Physik* 201.7 (Apr. 1865), pp. 353–400. DOI: 10.1002/andp.18652010702. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.18652010702>.
- [14] *Customizing code extraction - Help - LGTM*. URL: <https://lgtm.com/help/lgtm/customizing-code-extraction> (visited on 2019-03-21).
- [15] *CVE-2018-9139*. 2018. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9139>.
- [16] *CVE-2018-9264*. 2018. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9264>.
- [17] Ivan Bjerre Damgård. “A Design Principle for Hash Functions”. In: *Advances in Cryptology — CRYPTO’ 89 Proceedings*. Ed. by Gilles Brassard. New York, NY, USA: Springer New York, 1989, pp. 416–427. ISBN: 978-0-387-34805-6.
- [18] Yevgeniy Dodis et al. “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*. CCS '13. Berlin, Germany: ACM, 2013, pp. 647–658. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516653. URL: <http://doi.acm.org/10.1145/2508859.2516653>.
- [19] Peter G. Doyle. “Maybe there’s no such thing as a random sequence”. In: Mar. 2011.
  - [20] *Evaluation of random number generators*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik, 2013. URL: [https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS\\_20\\_AIS\\_31\\_Evaluation\\_of\\_random\\_number\\_generators\\_e.pdf](https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_20_AIS_31_Evaluation_of_random_number_generators_e.pdf).
  - [21] "dfranke" (Daniel Franke). *How I Hacked Hacker News (with arc security advisory)*. June 2009. URL: <https://news.ycombinator.com/item?id=639976>.
  - [22] Hans Freudenthal. “Realistic Models in Probability”. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by Imre Lakatos. Vol. 51. Utrecht University. 1986.
  - [23] *Gamblify | Slotmachines, Bookmaking, Online Casino*. URL: <https://www.gamblify.com/> (visited on 2019-03-17).
  - [24] Joel Gärtner. “Anaylsis of Entropy Usage in Random Number Generators”. MA thesis. Sweden: School of Cumputer Science and Communication, Sept. 2017.
  - [25] *Generating snapshot databases - Help - LGTM*. URL: <https://lgtm.com/help/lgtm/generate-database> (visited on 2019-03-20).
  - [26] *getrandom(2) - Linux manual page*. also available via `man 2 getrandom`. URL: <http://man7.org/linux/man-pages/man2/getrandom.2.html> (visited on 2019-03-17).
  - [27] *Godot Engine - Free and open source 2D and 3D game engine*. URL: <https://godotengine.org/> (visited on 2019-03-14).
  - [28] *Godot Engine Source Code*. version 3.1-stable. URL: <https://github.com/godotengine/godot> (visited on 2019-03-15).
  - [29] Ralph V. L. Hartley. “Transmission of Information”. In: *The Bell System Technical Journal* 7.3 (July 1928), pp. 535–563. DOI: 10.1002/j.1538-7305.1928.tb01236.x.
  - [30] Marcella Hastings, Joshua Fried, and Nadia Heninger. “Weak Keys Remain Widespread in Network Devices”. In: *Proceedings of the 2016 Internet Measurement Conference*. IMC '16. Santa Monica, California, USA: ACM, 2016, pp. 49–63. ISBN: 978-1-4503-4526-2. DOI: 10.1145/2987443.2987486. URL: <http://doi.acm.org/10.1145/2987443.2987486>.
  - [31] Nadia Heninger. *How not to generate random numbers*. June 2018. URL: <https://summerschool-croatia.cs.ru.nl/2018/slides/How%20not%20to%20generate%20random%20numbers.pdf>.
  - [32] Nadia Heninger et al. “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices”. In: *Proceedings of the 21st USENIX Security Symposium*. Vol. 8. Aug. 2012.
  - [33] *History - Bell Labs*. 2019. URL: <https://www.bell-labs.com/about/history-bell-labs/> (visited on 2019-02-24).
  - [34] *Introduction to data flow analysis in QL - Help - LGTM*. URL: <https://lgtm.com/help/ql/intro-to-data-flow> (visited on 2019-03-25).
  - [35] *Introduction to the buildsystem — Godot Engine latest documentation*. version 3.1-stable. URL: [https://godot.readthedocs.io/en/3.1/development/compiling/introduction\\_to\\_the\\_buildsystem.html](https://godot.readthedocs.io/en/3.1/development/compiling/introduction_to_the_buildsystem.html) (visited on 2019-03-21).
  - [36] *Introduction to the QL language — Learn QL*. URL: <https://help.semmle.com/QL/learn-ql/ql/introduction-to-ql.html> (visited on 2019-03-17).

- [37] John Kelsey. “The 90B Approach to Entropy Sources”. In: International Association for Cryptologic Research. Fault Diagnosis and Tolerance in Cryptography, 2018. URL: <http://conferenze.dei.polimi.it/FDTC18/shared/FDTC%202018%20-%20keynote.pdf>.
- [38] *lgTM.yml project configuration file - Help - LGTM*. URL: <https://lgTM.com/help/lgtm/lgtm.yml-configuration-file> (visited on 2019-03-20).
- [39] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: <http://doi.acm.org/10.1145/272991.272995>.
- [40] Ralph Charles Merkle. “Secrecy, Authentication, and Public Key Systems”. AAI8001972. PhD thesis. Stanford, CA, USA, July 1979.
- [41] Stephan Müller. *Documentation and Analysis of the Linux Random Number Generator*. Tech. rep. version 2.6. Bundesamt für Sicherheit in der Informationstechnik, Jan. 2019. URL: [https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG\\_EN.pdf](https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN.pdf).
- [42] John von Neumann. “Various Techniques Used in Connection With Random Digits”. In: *National Bureau of Standards applied mathematics series*. Ed. by USA National Bureau of Standards. U.S. Government Print Office, 1951.
- [43] Melissa E. O’Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA, USA: Harvey Mudd College, Sept. 2014. URL: <https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf>.
- [44] *OpenCV library*. URL: <https://opencv.org/> (visited on 2019-03-30).
- [45] *OpenCV source code*. URL: <https://github.com/opencv/opencv> (visited on 2019-03-31).
- [46] *OpenSSL Cryptography and SSL/TLS Toolkit*. URL: <https://www.openssl.org/> (visited on 2019-03-28).
- [47] *OpenSSL documentation for RAND*. URL: [https://www.openssl.org/docs/man1.1.1/man3/RAND\\_seed.html](https://www.openssl.org/docs/man1.1.1/man3/RAND_seed.html) (visited on 2019-03-27).
- [48] *PCG, A Family of Better Random Number Generators*. URL: <http://www.pcg-random.org/> (visited on 2019-03-28).
- [49] Christian Uldall Pedersen. email. B: Emails from Christian Uldall Pedersen. Mar. 2019.
- [50] *Profile - Hendrikto - LGTM community*. URL: <https://discuss.lgtm.com/u/Hendrikto/badges> (visited on 2019-04-01).
- [51] *Pseudo-random number generation - cppreference.com*. URL: <https://en.cppreference.com/w/cpp/numeric/random> (visited on 2019-03-27).
- [52] *Pseudo-Random Numbers (The GNU C Library)*. URL: [https://www.gnu.org/software/libc/manual/html\\_node/Pseudo\\_002dRandom-Numbers.html](https://www.gnu.org/software/libc/manual/html_node/Pseudo_002dRandom-Numbers.html) (visited on 2019-03-27).
- [53] *Python extraction - Help - LGTM*. URL: <https://lgTM.com/help/lgtm/python-extraction> (visited on 2019-03-21).
- [54] *Redis*. URL: <https://redis.io/> (visited on 2019-03-30).
- [55] *Redis Labs | Database for the Instant Experience*. URL: <https://redislabs.com/> (visited on 2019-03-30).
- [56] *Redis source code*. URL: <https://github.com/antirez/redis/> (visited on 2019-03-30).
- [57] Alfréd Rényi. “On Measures of Entropy and Information”. In: *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. Berkeley, California: University of California Press, 1961, pp. 547–561. URL: <https://projecteuclid.org/euclid.bsmsp/1200512181>.

- [58] *scikit-learn source code*. URL: <https://github.com/scikit-learn/scikit-learn> (visited on 2019-03-31).
- [59] *scikit-learn: machine learning in Python*. URL: <https://scikit-learn.org/> (visited on 2019-03-30).
- [60] *SCons: A software construction tool - SCons*. URL: <https://scons.org/> (visited on 2019-03-21).
- [61] *Semmler / About*. URL: <https://semmler.com/about> (visited on 2019-03-25).
- [62] Claude E. Shannon. “A Mathematical Theory of Communication”. In: *The Bell System Technical Journal* 27 (July 1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
- [63] "@3lbi0s" (Michal Stanek). *7-Zip Weak Encryption*. Jan. 2019. URL: <https://threadreaderapp.com/thread/1087848040583626753.html>.
- [64] Sebastiaan Terwijn. “The Mathematical Foundations of Randomness”. In: 2018.
- [65] *The GNU C Library*. URL: <https://www.gnu.org/software/libc/> (visited on 2019-03-28).
- [66] *The Linux Kernel Archives*. URL: <https://www.kernel.org/> (visited on 2019-03-02).
- [67] Myrion Tribus and Edward C. McIrvine. “Energy and Information”. In: *Scientific American* 224 (1971), pp. 179–188. DOI: 10.1038/scientificamerican0971-179.
- [68] Meltem Sönmez Turan et al. *NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation*. Tech. rep. National Institute for Standards and Technology, Jan. 2018.
- [69] *Using the PCG Library / PCG, A Better Random Number Generator*. URL: <http://www.pcg-random.org/using-pcg.html> (visited on 2019-03-27).
- [70] Bas C.M. Visser. “Additional source of entropy as a service in the Android user-space”. MA thesis. Radboud University, July 2015.
- [71] *Who is using scikit-learn?* URL: <https://scikit-learn.org/stable/testimonials/testimonials.html> (visited on 2019-03-30).
- [72] Taeill Yoo, Ju-Sung Kang, and Yongjin Yeom. “Recoverable Random Numbers in an Internet of Things Operating System”. In: *Entropy* 19.3 (2017). ISSN: 1099-4300. DOI: 10.3390/e19030113. URL: <http://www.mdpi.com/1099-4300/19/3/113>.