Randomness and Games on Ethereum

Master Thesis

Peter Allemann

University of Bern

Supervised by:
Prof. Dr. Christian Cachin
Luca Zanolini & Ignacio Amores Sesar

December 2021

Abstract

Randomness in computer systems serves many important use cases. Providing randomness to blockchains without compromising security or decentralization turns out to be no easy task. This work aims to provide an overview of currently used methods as well as an implementation to simulate a coin flip on Ethereum between two parties.

Acknowledgements

I would like to thank Prof. Dr. Christian Cachin for his time, patience, and guidance. Thanks also go to Luca Zanolini and Ignacio Amores Sesar for their counseling and insightful discussions. I would like to thank my parents who supported me throughout all these years that led me here. And finally, I am grateful for my girlfriend Nathalie Meichtry for her mental support and encouragement.

Contents

1 Introduction

Coming up with truly random numbers in a distributed environment like Ethereum is a well known problem [16]. The deterministic nature of Ethereum prevents the generation of truly random numbers. Yet, a variety of applications rely on randomness. Namely a lot of games often require some sort of randomness in order to be worthwhile playing. For example, any kind of card game is quite a dull experience if the player knows the order in which the cards are stacked in advance. However a dull experience is not the biggest problem. A dull game is probably not going to be played a lot anyway. A game that is potentially predictable by some of the participants or hosts is prone to being rigged or abused. In an ideal world we would be able to implement games on Ethereum that rely on some sort of randomness that is in no way or another affected by anybody. There are numerous ways how contracts try to come up with random numbers. Some of the more prominent ones might be the following, each with its advantages and disadvantages: Using the blockhash as a source of randomness, using a commit-reveal scheme, using an oracle as a provider of true randomness, using a collaborative scheme in which users work together to come up with a random number. It should be noted that there are several aspects that must be considered when trying to come up with random numbers. These aspects are randomness, security, cost and delay. Cost is defined by how much computational power is required to generate a pseudorandom number. In Ethereum every transaction has to be covered by an amount of gas, which can be payed for in Ether, according to the computational power that is required to execute it. The more computational power required, the more expensive the transaction becomes. Delay refers to the time it takes for the random number to become available. Randomness means that the produced random number should not be predictable by the consumer. Security entails that the produced random number is not only unpredictable to some but all users of the network, the produced random number should not rely on an external source in any way, the produced random number should not be abusable by the miners. Depending on the context in which the random number is required, these aspects should be considered appropriately. In other words, when choosing a method to come up with random numbers it should be considered what costs will be justifiable, how much time can pass until the random number is available, is the random number unpredictable by the users, is security at an appropriate level for what is at stake. Each of the before mentioned methods to come up with random numbers (blockhash, commit-reveal, oracle, collaboration) fail to completely satisfy all of these security criteria. Therefore, they should be considered unfit for applications in which there is a lot at stake.

Another way of coming up with pseudorandom numbers is with the use of a *verifiable random function (VRF)* [21, 25]. A verifiable random function consists of two main functions and requires at least two parties to participate. The first function, the *generator function G*, takes as input a *secret key SK* and an *alpha string* α, the output consists of the so called *beta string* β which serves as the pseudorandom number and the *pi string* π serving as a proof for correct execution. The first input, *SK*, is provided by the first party. The second input, α, is provided by the second party. The *alpha string* α could also be generated from the combined input of multiple parties. This allows for more than two parties to participate. The second function, the *validator-function V*, takes as input a *public key PK*, α and π; the output consists of a boolean value. If the value of said boolean is true, then *G* was executed with the correct corresponding *SK*. Since the validator function does not rely on *SK*, it can be executed by all participants. A *VRF* should satisfy the aforementioned security criteria. Implementation blueprints for such verifiable random functions are available. The VRF-solidity library [20] provides an implementation for an elliptic curve based validator function written in Solidity for Ethereum.

Since the VRF-solidity library [20] does not provide a generator function, this work provides an implementation for an elliptic curve based generator function in Solidity as well as in the Go programming language. The *pi strings* generated by these functions can be verified by the VRF-solidity library. Further it provides an implementation for an RSA based generator and validator function, also written in Solidity and in Go. Similar to elliptic curve multiplication, where there exists no efficient inverse operation, the RSA based approach relies on the fact that there is no efficient way of factoring large integer numbers. These two different methods are then analyzed and compared in regard of cost, delay, randomness and security.

1.1 Goal and Motivations

Providing true randomness or pseudorandomness in a blockchain like environment like Ethereum is not trivial. There is a variety of applications that rely on randomness. The most obvious of which are probably games which rely on some sort of hidden information (e.g. card games, lotteries, etc.). It should be obvious that true randomness can never be derived from a deterministic system and therefore the only way to get access to true randomness is through an oracle that feeds information to the blockchain from the outside. The caveat is, of course, that such an approach undermines one of the primary properties of a blockchain, and that is being a decentralized and highly redundant system. Relying on a single entity, like an oracle, introduces a single centralized point to the system (an argument for a decentralized system that provides true randomness to the blockchain could be made, but even then there is no easy way to guarantee that the true randomness was generated in a sound manner and without malicious intent). Providing pseudorandomness to the blockchain still requires some external information but there are some simple ways to guarantee, or at least reinforce, that the involved parties have a hard time guessing the outcome in advance.

Here it should be noted that there is a significant difference between an application that relies on pseudorandomness in order to solve outcomes between opposing players and an application which allows players to play against the 'house' (like in roulette).

An easy solution for two players to settle a bet, is to let both players first commit to a secret number s_i by announcing its hash *hash*(s_i) publicly. After a predefined time (number of new blocks mined) the players reveal their secret number s_i. Now each player can verify *hash*(s^i). Once both players are satisfied that the hashes are indeed correct, the hash of the concatenated numbers *hash*($s_1 \| s_2$) can be used as a pseudorandom number. This is of course assuming both players act in a sound manner (no censorship attacks against each other). A smart contract can guarantee that both players must follow the protocol. If more players are involved, this becomes more complex because the last player (a player could also enter multiple times) to reveal his secret number s_i has an advantage over the others (Byzantine fault).

An application which allows players to play against a 'house' is more complex, since a smart contract has no way of storing information outside the blockchain by itself. It has to either rely on an *externally owned account* (EOA) to handle hidden variables used in a commit-reveal scheme, or rely on multiple competing players to come up with the pseudorandomness which brings the same difficulties as mentioned in the paragraph above.

The goal of this thesis is to provide an overview of the currently available methods to generate pseudorandomness as well as providing a sound implementation for a simulation of a bet on a coin flip between two parties in a blockchain environment like Ethereum using a verifiable random function (VRF). The benefit of using a VRF over a commit-reveal scheme being that the party owning the secret key does not have to ever reveal its secret key and therefore possibly has to participate in less communication.

2
The Problem With Randomness

Randomness plays an important role in science, art, statistics, cryptography, gaming, gambling, and other areas. Because of that, there are of course also a lot of applications that rely on randomness. However, providing randomness to applications, or computer systems in general, is not a trivial task. Various approaches for providing random numbers are being implemented and used currently, a few of them are being discussed in the next chapter. They provide either true random numbers or pseudorandom numbers. None of these approaches are perfectly safe however. For some applications, like a lottery for example, security is a major concern while for others it is not. Depending on the method used they also come with different costs attached.

2.1 Randomness in Computer Systems

Achieving true randomness in computer systems is hard. This is especially true for a deterministic system like Ethereum, in which a consensus has to be reached among a majority of nodes. This implies that true randomness is impossible to achieve within Ethereum [16]. Otherwise each node would come to a different result when evaluating transactions. Yet there is a real need for random numbers in Ethereum applications. Especially in games.

2.2 True Randomness Versus Pseudorandomness

There are various phenomenons in nature that can be considered as truly random. Meaning, these phenomenons seem to lack to follow a predictable pattern. Even if the probability distribution of a certain event is known and therefore the frequency of an outcome in multiple observations can be predicted, the outcome of a single observation is by definition unpredictable. However, the deterministic nature of Ethereum commands that we cannot rely on true random numbers from within the Ethereum network. True random numbers can only be acquired from external sources through so called oracles [16]. There are a number of options how such external sources could provide random numbers. For example, by observation of cosmic background radiation, atmospheric noise or the decay of radioactive material. Randomness acquired in such a manner can be considered truly random, since it is impossible to predict the outcome in advance. However, relying on an external source as a provider for true random numbers has its downsides. It requires that the external source is trusted to act with sound intent. Furthermore it requires that the system now relies on an external source as a single centralized entity. An alternative option is pseudorandomness. The term pseudorandomness measures the extent to which a sequence of numbers appear to be random to an outside observer, even though produced by a completely deterministic, and therefore repeatable, algorithm. This can be achieved by hiding either the algorithm or parts of the algorithm to the observer of the random number. Here it should be noted that Chainlink [10] (which is discussed in detail in subsection 4.2.4) actually provides a solution to pseudorandom numbers in a decentralized manner. Further, pseudorandom numbers always come with a delay because they have to be calculated in some way or another while true random numbers have to be provided from an outside source and can be available the soonest within the next new block.

2.2.1 The Cost of Pseudorandomness

Since pseudorandom numbers have to be calculated by an algorithm, they come with a computational cost attached. There are various methods in order to come up with pseudorandom numbers. Depending on the algorithm used, these computational costs vary greatly. In Ethereum transactions have to be payed for in gas [16]. Gas can be purchased with Ether. The more computationally expensive a transaction is, the more gas is required. Because of that circumstance, the method which provides pseudorandom numbers to a contract should be chosen appropriately. It does not make sense to use a method that is very sophisticated but generates a tremendous amount of computational costs, if there is not a lot at stake in the first place.

2.3 Integrity of Random Numbers in a Blockchain Environment

One of the challenges when trying to provide randomness or pseudorandomness to a blockchain like Ethereum is making sure no one is being able to abuse it with malicious intent. A provided random number or pseudorandom number should not be predictable or manipulated by anyone that consumes or relies on its outcome. Further, the provider should not be able to manipulate the outcome once the random number has been generated. There might always be multiple entities that try to abuse the system for their personal gain. There are at least three main parties that should always be taken into consideration when analyzing the integrity of a provided random number or pseudorandom number. First off, the provider of the random number or pseudorandom number. If the random number or pseudorandom number is being generated by the same source that is consuming it later, that source usually has a strong incentive to manipulate the provided randomness to his own advantage. Because of this, the provider of randomness should either be decoupled from the consumer or provide proof that the randomness was generated in a sound manner. This can be achieved by the use of verifiable random functions. For obvious reasons the consumer of randomness has every bit of incentive to manipulate the randomness or pseudorandomness to his own favor. The consumer should be decoupled from the process of creation of randomness if possible. Lastly, the block miners. The block miners eventually decide which transactions are being included in the blockchain. Because of this, they are naturally in a powerful position to manipulate when and whether events take place on the blockchain. Further any collaborations between these three parties should also be taken into account. It can never be guaranteed that the integrity of a random number or pseudorandom number can be completely trusted. The process in which it has been generated however, should be structured in a way that incentivises the randomness provider to act in a sound manner while at the same time make it as difficult as possible for the consumer and for miners to affect the outcome of the randomness or the way the randomness is being consumed by the consumers.

2.3.1 Blockhash as a Source of Pseudorandomness

Using the blockhash as a source of pseudorandomness [5, 28] allows the block miner to decide whether he wants to publish the found block, or if the resulting pseudorandomness does not work in their favour, to discard it. Obviously the potential gain of discarding the block should out weight the loss the miner would experience by broadcasting the found block. It has been tried to use a combination of two different block hashes as the source of randomness in an attempt to circumvent this problem [9]. By doing so usually two predetermined blocks of a given height are being used. For example block number `h` for the first block and block number `h+5` for the second block. However, this approach is rather short-sighted because it does not guarantee that these two blocks are coming from two separate miners. Further, if the blocks are being found by two different miners, it does not prevent the miners from collaborating together. But most important of all, the miner of the second block (block number `h+5`) does not even have to collaborate with the miner of the first block. The blockhash of the first block (block number `h`) is available and visible to anyone anyway. In this sense, even if there had been more than two miners, the miner of the last block alone can decide or at least influence (by not releasing the found block) the outcome of the attempt at providing pseudorandomness [9].

2.3.2 Randomness in Games on a Blockchain

Randomness is an integral part of many games and is often the reason why these games are interesting or even worth playing. Secure randomness is a property that is difficult to obtain on blockchains. Blockchain based gaming is promising in that it allows players to truly own their in game assets compared to conventional game models where players are relying on the game provider to administer their assets. Randomness could be used to determine player rewards for such games. Through Non Fungible Tokens (NFT) a player could gain provable ownership of his in game assets and make sure that they are immutable. Obtained assets could even be used in different blockchain based games which would incentivize a market for such assets. Today such a market already exists and its assets are quickly gaining value [6]. In order to provide randomness to a blockchain based game, developers have to decide whether they want to rely on centralized off-chain solutions or make us of an on-chain pseudorandomness algorithm. Off-chain solutions often come with the drawback of being centralized (one exception being Chainlink [10], which is discussed in more detail in subsection 4.2.4) and non-transparent, while on-chain pseudorandomness algorithms are often open for a variety of attack vectors (censorship attack, collaboration with a miner, forking, unsound implementation [9]). With market assets gaining more value, the requirements for secure and transparent randomness is increasing rapidly [8].

3
Background

This chapter covers the theoretical and technical background of using and providing pseudorandom numbers in distributed systems.

3.1 Blockchain

A blockchain is a system which allows to record information in such a way that makes it very difficult for these recordings to being changed.

In general a blockchain is a digital ledger which contains a number of blocks each containing a number of individual transactions. This digital ledger or blockchain is duplicated and distributed to each node of the network. Once a new transaction is made, it will be broadcasted to the network and upon agreement among participants, is accepted and included in the blockchain as part of the next block. The network considers the longest blockchain as the current valid blockchain. Each block contains the hash value of its predecessor. This mechanism makes it particularly difficult to change the contents of the blockchain.

Not all blockchains are equal. A blockchain can be implemented in a variety of ways and with a wide spectrum of properties. The following are usual components of a blockchain [16]:

- A peer-to-peer network propagating transactions and blocks.
- Messages or Transactions.

- A consensus protocol.
- A state machine processing transactions according to consensus rules.
- A chain of cryptographically secured blocks.
- A game-theoretically sound incentivization scheme (e.g. proof-of-work)
- A software implementation of the above.

3.1.1 Ethereum

Ethereum is sometimes described as 'a world computer' and is an open-source project. Ethereum is a decentralized computer system that executes so called *smart contracts* as its programs. It uses a blockchain to synchronize and store the systems state. The cryptocurrency *Ether* largely governs state changes to the system. Ethereum allows to build decentralized applications that are transparent and are robust against censorship [16].

3.2 Smart Contracts in Solidity

Solidity is a high-level object-oriented programming language designed to implement smart contracts on the Ethereum Virtual Machine (EVM). It is influenced by C++, Python and JavaScript [15]. In Ethereum there are two types of accounts, both types can be identified with an address. Externally owned accounts (EOAs) are controlled by users and do not contain any other data than the currently available amount of Ether. A public- and private-keypair allows to make transactions from these accounts, whereas a transaction can only be made if it has been cryptographically signed with the corresponding private key. Contract accounts on the other hand are governed not by users but by their code that is executed by the EVM. This code is what is referred to as a *smart contract*. Smart contracts are simply computer programs that run deterministically on the EVM. The word contract has no legal implications. Since the contents of the blockchain are immutable, the code of a smart contract cannot be changed. This means, the only way to change a contract is to deploy a new updated version of it that lives in parallel to the old one. However, a smart contract can be deleted by its owner in order to release the occupied address space (incentivized by negative Gas costs). The outcome of an execution of a smart contract depends only on the current state of the Ethereum blockchain, no matter who runs it. A smart contract has access to its own state as well as to the transaction that called it and some basic information about the most recent blocks. A smart contract only reacts to a transaction started from an EOA. It can call other smart contracts, but it will never start acting without an impulse from an EOA. Transactions are atomic, meaning each transaction is executed completely and in order. There are no parallel executions of smart contracts. If a transaction fails it will not be included in the blockchain, treating it as if it was never called in the first place [16].

3.3 Entropy

Entropy is the level of uncertainty or surprise in a possible outcome of an event. Shannon entropy [26] is defined as follows:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \cdot \log_2 P(x_i)$$

X is a discrete random variable that can take on the values $x_1,...,x_n$. The probability for X to take on the value x_i is $P(x_i)$. Since we want to measure the entropy in bits, $\log_2$ is used. $H(X)$ is the resulting entropy in bits. Generally as the possible number of values of X increase, the provided entropy increases as well but not as fast as the number of possible values. Uniform distribution of outcomes maximizes entropy. A derivation from uniform distribution of outcomes will always lower entropy.

A fair coin will result in one bit of entropy per coin toss:

$$\begin{aligned} H(\textit{Fair coin toss}) &= -\sum_{i=1}^{2} \frac{1}{2} \cdot \log_2 \frac{1}{2} \\ &= -\frac{1}{2}(-1) - \frac{1}{2}(-1) \\ &= 1 \end{aligned}$$

A biased coin ($P(x_1) = 0.8$) will result in less than one bit of entropy per coin toss:

$$\begin{aligned} H(\textit{Biased coin toss}) &= -\sum_{i=1}^{2} P(x_i) \cdot \log_2 P(x_i) \\ &= -0.8 \cdot \log_2(0.8) - 0.2 \cdot \log_2(0.2) \\ &= 0.7219280948873623 \end{aligned}$$

Throwing a fair eight-sided die will result in three bits of entropy:

$$\begin{aligned} H(\textit{8 sided die}) &= -\sum_{i=1}^{8} P(x_i) \cdot \log_2 P(x_i) \\ &= -8 \cdot 0.125 \cdot \log_2(0.125) \\ &= 3 \end{aligned}$$

Entropy can be used to roughly estimate the measure of unpredictability of a cryptographic key. For example, a 256-bit long key that was generated through a series of 256 fair coin tosses takes on average 2^{255} guesses to break by brute force [5]. Since all transactions on the Ethereum blockchain are deterministic, the outcome of each transaction has foreseeable and calculatable outcome that involves no uncertainty at all.

3.4 True Randomness

Randomness is commonly viewed as the property of a process that results in events which are unpredictable in advance. That is to say independent from a given starting point or setting, the outcome of the process cannot be predicted in advance. Even if full knowledge of the initial setting and workings of the involved mechanisms and infinite computational power are available. Knowledge about the distribution of multiple events also does not provide any insights into the outcome of the next event. For example, a fair coin is a coin which is equally likely to land on heads as it is to land on tails if it is being tossed into the air. If a fair coin is being flipped repeatedly, the ratio of landing on tails vs heads will gravitate towards a one to one ratio if enough flips are being made. But the knowledge about the frequency of an outcome must still not grant any insight into the outcome of any one individual event. If said fair coin lands heads for the tenth time in a row, it will not be more or less likely to land heads on the eleventh flip. The chances of each outcome will remain the same, independent of previous outcomes. This is sometimes called true randomness. True randomness is nondeterministic and aperiodic, meaning there is virtually no way of reproducing an event reliably.

True randomness is often used due to its inherent fairness. Applications of true randomness can be observed in many different fields. Probably the most obvious application that comes to mind is gambling that includes dice or some other source of true randomness like a roulette wheel. Another prominent application for true randomness are lotteries. True randomness also plays an important role in the generation of data encryption keys [16].

Here it should be noted that according to logician Frank P. Ramsey pure true randomness is impossible, due to the fact that any structure will necessarily contain an orderly substructure [22]. Given this impossibility however, an effort in studying the various degrees, and therefore various qualities, of randomness can certainly be made [18].

For the rest of this work, true randomness refers to an event that is unconditionally unpredictable by an adversary, even if he possesses infinite computing resources.

3.4.1 General Solution for Creating True Random Bits

In order to create a string of truly random bits the following steps should be taken. First, one has to gather some bits from a source that is unknown to the adversary. Even though these bits might not necessarily all be independent, they must contain some information that is unavailable to the adversary. Second, it has to be determined how many bits were gathered that are independent and not guessable by the adversary. The amount of these bits is referred to as entropy. Third, with the help of a hash function an output can be created that is functionally dependant of all input bits, while all output bits are functionally independent from each other. Hash functions which are considered cryptographically strong (e.g. Keccak-256) are regarded to posses these properties [16]. The output can now be considered as a set of independent and unguessable bits that can be used with confidence as random bits [5].

3.4.2 Practical Solutions to Generate Randomness

True randomness can be created in two different ways.

First, it can be derived from observing the environment. Following are a few examples how true randomness can be obtained. Brownian motion [3] describes the movement of larger particles that are interacting with lighter and faster moving molecules and can be observed via microscope. Random.org [23] uses radio receivers to pick up atmospheric noise in order to generate randomness. HotBits [29] is observing time intervals between beta decay of radioactive elements. Repeated coin flips will generate one bit of randomness at a time, which is not a very efficient but a very simple method. Arnold G. Reinhold proposes a dice-ware generator that uses a few dice and a shoe box to create high-quality true randomness [7].

Second, true randomness can come from systems whose behaviour is sensitive to small variations of its starting condition. An example of such a system is the double pendulum. A double pendulum is a pendulum that has another pendulum attached to its end. The motion of such a double pendulum is chaotic in nature and very sensitive to its initial conditions. Depending on how accurate the initial conditions are known, the longer the motion of the pendulum can be predicted. However, no matter how accurate the initial conditions are known, after a certain point the motion of the double pendulum becomes completely unpredictable [27].

As long as one is careful not to introduce unwanted patterns into the generation process by accident, the possibilities to come up with true random numbers are virtually endless.

3.5 Pseudorandomness

Pseudorandomness is derived from algorithms which try to immitate true randomness. The main goal of a pseudorandom number generator (PRNG) is to produce an output that is indistinguishable from true randomness to an outside observer that has no information about the seed used in conjunction with the algorithm. This means, the algorithm has to produce an output that is apparently free of any patterns or regularities. Further, the knowledge about previously produced outputs must not aide the adversary to guess the next output. Pseudorandom number generators are deterministic and periodic, which means an outcome can be reproduced if the right starting parameters are known. Also, only a limited amount of different outputs can be created. One of the advantages pseudorandom number generators have over true random number generators is their efficiency at which number can be output.

Due to the above stated properties one prominent applications of pseudorandomness are computer simulation and modelling.

For the rest of this work, pseudorandomness refers to an event that is unconditionally unpredictable by an adversary, only if he possesses limited computing resources.

3.5.1 The Seed

In case a source for true random numbers is available but does not provide a enough random bits, a PRNG can be used to increase the number of random bits to an outside observer. The PRNG uses the true random bits as an input, a so called seed, and calculates the corresponding output. Such a PRNG should be cryptographically strong, meaning that it has been proved that unless the knowledge about the seed is known to the adversary he is only able to guess the next output bit with a likely hood no more than $\frac{1}{2} + \epsilon$, where ϵ decreases exponentially with the length of the seed (under the assumption P$\neq$NP) [24]. So the length of the seed determines how robust the produced output is against a brute force attack. However, this is only feasible against an adversary with limited computational power [5]. As a practical example, there exists software which initializes the random number generator of the operating system by using user generated input (for example movement of the mouse cursor).

3.6 Randomness and Its Role in Key Generation

In cryptography one of the most common needs for random values is the generation of cryptographic keys (or passwords). To illustrate lets consider the following example. In order to crack a four-letter-password one would theoretically need to try every single possibility of four-letter combinations (worst case). Which would result in 26^4=456976 attempts. However, if the password was created by a user, it is highly likely that it is in the form of a four-letter word that is easy to remember. This circumstance drastically lowers the possible candidates for the password. There are roughly 5500 four-letter words in the English dictionary [1], so in order to crack the four-letter password only about one hundredth of all possible combinations have to be tried. So there is a need for values which cannot be guessed by an adversary any more efficient than by trying every single available possibility [5]. The easiest way to guarantee this property for a cryptographic key, is to rely on some secure source of randomness for the generation of said key.

In Ethereum each Externally Owned Account (EOA) is being controlled by a private key. The corresponding public key can be derived via elliptic curve multiplication from the private key. The address of the EOA is then derived from part of the public key. The private key is essentially a number between 1 and 2^{256} (actually 2^{218} since the first 38 digits are fixed due to the order of the elliptic curve). In order to create a new EOA, a private key from this tremendously large range of numbers has to be chosen at random.

3.7 Public Key Cryptography

Public key cryptography makes use of unique keys to secure information. Such keys can be obtained by mathematical functions which are easy to compute but difficult to invert (sometimes called one-way-functions or trapdoor functions). The inverse of such a function is in general only obtainable by use of brute force methods. For example, it is very simple to multiply large prime numbers. But the inverse, factorizing a product of two large prime numbers is not so simple. The larger the prime numbers used, the more computationally expensive this operation gets. Figuring out the prime factors of the number 8469277 would involve a lot of trial and error until the first prime is found. While multiplying 1997 and 4241, which results in 8469277, is a simple operation.

3.7.1 Elliptic Curve Multiplication

With newer algorithms like Quadratic Sieve [12] the amount of work required to factorize large numbers actually decreases relative to the length of the number. This implies that in order to keep up with the available computational resources larger and larger prime numbers have to be used. At the same time the amount of work to multiply such large numbers relative to the amount of work to factorize them becomes smaller and smaller. This situation is not sustainable in the long term [11].

In Ethereum key pairs are generated via elliptic curve multiplication. Among further applications are digital signatures and pseudorandom generators. Elliptic curve cryptography allows to use smaller keys compared to non elliptic curve cryptography while providing equivalent security [11]. Elliptic curve multiplication is a function that is practically irreversible. To this day no algorithm has been found that improves upon the naive approach of simply trying each possibility in order to reverse an elliptic curve multiplication [11]. The elliptic curve used by Ethereum (and also by Bitcoin) is defined as a set of points *(x,y)* which satisfy the equation $y^2 \mod p = (x^3 + 7) \mod p$. The parameter *p* is a very large prime number (*p* = $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$). This curve is called *secp256k1* and is defined in Standards for Efficient Cryptography

(SEC 2) [17]. In order to derive a public key from a private key, the multiplication *K* = *k* * *G* has to be performed. Here *K* is the public key and consists of the point *(x,y)* which satisfies the above mentioned solution $y^2 \mod p = (x^3 + 7) \mod p$. The private key is represented by *k* and is a scalar. *G* is a predefined point. The elliptic curve multiplication works a little different from regular multiplication. Regular multiplication can be seen as a series of regular additions ($2+2+2+2+2 = 5 \cdot 2$). Elliptic curve multiplication works in a similar way. Other than regular addition of two points, elliptic curve addition is defined in the following manner. Consider the equation $P_3 = P_1 + P_2$. Drawing a straight line between P_1 and P_2 will cross the elliptic curve in exactly one Point P_3'. Now all that is left to do, is to reflect this point at the x-axis which will finally bring us to the point P_3. Similar to regular multiplication, elliptic curve multiplication is simply a consecutive execution of additions. There are some special cases defined, if P_1 and P_2 are the same points, the tangent from P_1 to the elliptic curve is being used to derive the next point on the curve, further if $P_1 = (x,y)$ and $P_2 = (x,-y)$ then there won't be any intersection between the elliptic curve and the line between the two points (in this case P_3 is simply defined as P_1). A public key is represented as a series of 130 hexadecimal characters and consists of a prefix (0x04) followed by the x-coordinate (64 hex) followed by a y-coordinate (64 hex). There is no known approach which can invert a public key and will find its corresponding private key. The only way to do so would be to multiply every possible private key with the generator point and compare the result with the private key. This approach would take quite a while. Try iterating through 2^{256} (continue reading when you are done).

3.8 Hash Functions

A hash function generally takes an input of arbitrary length and maps it to a fixed length output. Due to the fact that the input space is larger than the output space, such hash functions are many to one functions by nature. This means, many different outputs can generate the same output value, also called a collision. For the hash functions used by Ethereum (e.g. Keccak-256) it is virtually impossible to find such collisions due to the smooth distribution of the output over an enormous output space. A cryptographic hash function, is a function which satisfies the following properties. Determinism: The same input value will always generate the same output value. Verifiability: It is efficient to calculate the hash (linear complexity). Noncorrelation: If only a single bit in the input value is being changed, the output value must change in such a way that it is impossible to correlate it in any way to the original input value. Irreversibility: The only way to reverse the hash function is by brute force (trying out every possible input). Collision protection: It should be practically impossible to find two different input values that correspond to the same hash output value.

Cryptographic hash functions can be used e.g. in order to sign messages (fingerprint), as part of pseudorandom number generators or in commit reveal mechanisms. In Ethereum they allow to derive addresses from public keys and are a key component of proof of work [16].

3.8.1 Keccak-256

The Keccak-256 hash function is used in Ethereum quite often. It is also used to derive an Ethereum address from a public key. In order to do so Keccak-256 is being used to hash the public key *K* like so `Keccak256(PK)`. The last twenty bytes (least significant bytes) of this hash output refer to the corresponding Ethereum address.

The following is an example of how to derive the address of an EOA from its private key *k* by using elliptic curve multiplication and the hash function Keccak256. Given the private key *k*:

```
k = 0xddde823cc26fb3cd14b59da1977622016d506d1030810a6f
    02e719045ac7d30a
```

Public key *K* can be derived by elliptic curve multiplication of the private key *k* with the generator point *G*:

```
K = 0xddde823cc26fb3cd14b59da1977622016d506d1030810a6f
    02e719045ac7d30a * G
```

Like the generator point *G*, the public key *K* consists of an x- and y-coordinate:

```
K = (x, y)
x = 0x911530718d23180b83d32e69a58094f8272716b3760b1a47
    273b60bcd7ce7df5
y = 0x2f83f720f08b2128a023303bbe5be8e269598fad06b1c7b9
    846851e24dd67b91
```

The public key *K* is represented by concatenating its x- and y-coordinates like so:

```
K = 911530718d23180b83d32e69a58094f8272716b3760b1a4727
    3b60bcd7ce7df52f83f720f08b2128a023303bbe5be8e26959
    8fad06b1c7b9846851e24dd67b91
```

Now the hash of the public key *K* can be calculated:

```
Keccak256(K) = f06eafa160da4630dd6d683e4de2425af9767d7
               c99091cb149cdf5c9bb3ea77a
```

By taking the twenty least significant bits of this hash value we get the address of the corresponding EOA:

```
Address = 4de2425af9767d7c99091cb149cdf5c9bb3ea77a
```

3.9 Digital Signatures

Digital signatures can be used in order to guarantee that a message was actually sent by the sender himself and was not altered in the transmission. This can be accomplished by adding the value *sign(message, secret key)* to the message. Anyone who now wishes to verify that the message was indeed sent by the owner of the *secret key* and was not altered in the process of transmission can simply execute *verify(message, public key, sign(message, secret key))* which will only return the value *True* if the proper parameters were used.

Ethereum transactions are signed with the private key belonging to the Externally Owned Account (EOA) which initiated the transaction. Elliptic curve mathematics ensure that everyone can verify that the transaction was actually initiated by the owner of the private key without actually revealing anything about the private key itself.

3.10 Probabilistic Algorithm

A probabilistic algorithm assigns elements from one set to elements from another set according to its innate probability distribution. It can be seen as a function $F : X \times Y \rightarrow [0, 1]$ that has to satisfy the condition

$$\forall x \in X : \sum_{y \in Y} F(x,y) \leq 1$$

So if $X = Y = \{1, 2, 3\}$ and $f(2, 3) = \{(1, 1) \mapsto 0.5, (1, 2) \mapsto 0.25, (1, 3) \mapsto 0.25, anythingelse \mapsto 0\}$ then calling f(2,3) will return (1,1) half of the time, on the other half it will return (1,2) or (1,3) with equal probability.

3.11 Formalization of a Verifiable Random Function

The general idea of a verifiable random function is to create pseudorandomness through a function f_s. The subscript s denotes a secret seed. The knowledge of s allows to evaluate the function f_s at point α as well as the construction of a proof π. π alone can be used to verify that the value $f_s(\alpha)$ is indeed correct. At the same time, π must not compromise the unpredictability at any other point at which the function f_s will be evaluated and no corresponding π is available.

The following is an attempt at defining VRFs with as little mathematical notation involved as possible. A rigorous mathematical definition of VRFs can be found in the paper *Verifiable Random Functions* by Micali, Rabin and Vadhan [25].

3.11.1 Generator, Evaluator and Verifier

In order to construct a VRF three main algorithms are required, each of which are polynomial in time in regard to their input. This means, each algorithm will terminate after a number of steps which can be described as a polynomial of its input length for any given input.

The first of these three algorithms is the *function generator* called *G*. It takes a *security parameter k* as input from which it produces a keypair *public key PK* and *secret key SK*. The *security parameter k* is in the form of a unary string, which is a string consisting of the same symbol repeated for *k* times. *G* has to be probabilistic. Meaning, each different *security parameter k* is mapped to a different keypair *SK* and *PK* at random. *SK* and *PK* are binary strings. The *function generator G* does not calculate *PK* from *SK*, it merely chooses a keypair from a given *k*.

The second of these algorithms is the *function evaluator* $F = (F_1, F_2)$. The *function evaluator F* takes two binary strings as input parameters. The first of which is *SK* and the second α. The output of *F* is two binary strings, the $\beta = F_1(SK,\alpha)$ and $\pi = F_2(SK,\alpha)$. F_1 and F_2 have to be deterministic. Meaning, for a given input the algorithm will always arrive at the same output while always going through the exact same sequences.

The third and last of these algorithms is the *function verifier V*. It takes as an input the four binary strings *PK*, α, β and π. The *function verifier V* is probabilistic and its output is either *true* or *false*.

3.11.2 Property Requirements of a VRF

Assume any three functions $a : \mathbb{N} \rightarrow \mathbb{N} \cup \{*\}$ and $b, s : \mathbb{N} \rightarrow \mathbb{N}$ that are all computable in polynomial time with regard to *k*. Further *a(k)* and *b(k)* both have to be bounded by a polynomial in *k*. This means there exists a function *g(k)* which takes a smaller or equal value for any *k* than *a(k)* and *b(k)* as well as that there exists a function *h(k)* which takes a greater or equal value for any *k* than *a(k)* and *b(k)*. In case *a(k)* takes on the value * it does not have to be bounded by a polynomial in *k*. It can be said that (*G, F, V*) is a *verifiable pseudorandom function (VRF)* of input length *a(k)*, output length *b(k)* and *security s(k)* if the following three properties are satisfied:

3.11.2.1 Domain-Range Correctness and Complete Provability

First, the algorithm F_1 has to produce for each input α of length *a(k)* a corresponding output β of length *b(k)*. Further, the algorithm F_2 has to produce, from any input α, a corresponding output π which will then result in an output *true* when used as input for the *function verifier V*. The aforementioned properties have to hold with an increasing probability ($1\text{-}2^{-\omega(k)}$) with respect to the amount of assigned keypairs in algorithm *G*. In other words, any input (*SK*, α) of correct length to the *function evaluator F* produces an output (β, π) of correct length which can then be used as as an input (*PK*, α, β, π) to the *function verifier V* to generate *true* as an output. If this was not the case, the parameter α could be chosen in a way so that the *function evaluator F* would produce outputs which could never be proven to be correct by the *function verifier V*.

3.11.2.2 Unique Provability

Second, given the parameters *PK*, α, β_1, β_2, π_1, π_2 where β_1 and β_2 are not equal, the *function verifier V* will only output *true* for one of the two proofs provided. In other words, for any point *(SK*,α*)* at which the *function evaluator F* is being evaluated, there must be exactly one corresponding output (β, π) which will produce *true* as an output when used as input to the *function verifier V*. If this was not the case, and two (or more) different pairs of parameters (β_1, π_1) and (β_2, π_2) both can be used to validate the input *(SK*, α*)*, the executor of *function evaluator F* could choose to publish the output parameters which are favorable for him.

3.11.2.3 Residual Pseudorandomness

Third, an adversary who is allowed to query the *function evaluator F* and *function verifier V* for *s(k)* steps and who has access to *(SK,PK)* but not to α must not be able to reproduce β = *F(SK*,α*)* with a probability greater than $\frac{1}{2^{|\beta|}} + \frac{1}{s(k)}$ where $|\beta|$ is the length of the string β. This means that a given output of $\beta = F_1(SK,\alpha)$ should be indistinguishable from random to anyone who does not have access to α.

If these three properties hold, $F_1(SK,\cdot)$ can be seen as a VRF.

4 Common Practical Solutions and Related Work

In this chapter common practical solutions for coming up with random numbers and pseudorandom numbers on Ethereum are discussed.

4.1 Common Methods to Provide True Random Numbers

The following is an example for how contracts can get access to true random numbers on the Ethereum blockchain.

4.1.1 The Oracle Pattern

One prominent way of coming up with true randomness is by relying on an outside service through a so called oracle. The idea is that some external entity has access to a reliable source of randomness like observing cosmic background radiation, atmospheric noise or the decay rate of radio active elements. A contract can request a true random number from an oracle. The oracle then interacts with the external entity and provides the true randomness to the contract. This approach however, undermines the basic idea of the blockchain as a decentralized system, since the provider of the randomness represents a single central entity. In other words, the application which relies on an oracle as a source of randomness has to blindly trust said oracle as well as the data provider. There is no easy way to ensure that the provided numbers are truly random. Even worse, if there is Ether at stake, there is no way of ensuring that the oracle and the data provider are acting without malicious intent. This approach can provide both true randomness as well as pseudorandomness. The external provider will most likely rise a fee for a request of a true random number or pseudorandom number. A further fee will probably be raised by the oracle as well. In a best case scenario the requested true randomness or pseudorandomness can be expected to be available within a delay of one block. In order to decrease the trust required in a single oracle a request could be posed to multiple oracles. The provided results are then compared. A result is only accepted if a given subset of said oracles provide the same result. This would of course increase the costs by a multitude of oracles queried.

4.2 Common Methods to Provide Pseudorandom Numbers

The following are common examples contracts try to generate pseudo random numbers on the Ethereum blockchain.

4.2.1 Blockhash as a Source of Randomness

One simple solution many contracts make use of, is to use the latest blockhash as a source of pseudorandomness. This can be achieved by the following simple code fragment.

```
// Supposedly pseudorandom number generator.
function newPseudoRandomNumber() internal view returns
(uint) {
    return uint(blockhash(block.number-1));
}
```

The problem with this approach is that `block.number` is of course readily available on the blockchain. Because of that, anyone who wishes to predict that supposedly pseudo random number can do so by calling `uint(blockhash(block.number-1));`. Further a miner could choose to withhold a found block if the outcome will be to his disadvantage. However, this is only a relevant concern if the resulting disadvantage is comparable to the block reward of the withheld block. This method comes with no additional costs attached.

4.2.2 Commit-Reveal

The commit-reveal scheme is yet another cost-efficient way of providing pseudo random numbers. It represents a substantial improvement over the use of the blockhash alone with little costs attached. In this approach a trusted party, chosen by the contrast, provides a seed which will be hashed together with the blockhash of a future block. The provided seed will be sealed by hashing it together with the address of the trusted party. The sealed hash is then stored in the contract that requires the pseudorandom number. The actual seed remains hidden until the previously defined blockhash becomes available. Once the previously defined blockhash becomes available, the seed can be revealed by the trusted party. With help of the sealed hash it can now be verified that the provided seed has not been altered since. The pseudorandom number is now generated by hashing the seed together with the blockhash. This way the pseudo random number can no longer be easily predicted by use of `block.number`. Further miners can no longer gain any insight to what the generated pseudo random number will be based on the blockhash alone. In order to predict the outcome of the pseudorandom number the seed provider would have to collaborate together with the miner of the predefined block. Since hashing can be considered relatively inexpensive, this method comes with very little extra cost attached. But the pseudorandom number is not available immediately. The pseudorandom number can be expected to be available the soonest after two blocks. However, it still requires to trust the seed provider to reveal his seed once the predefined block has been mined. Since the seed provider has access to the outcome of the pseudorandom number as soon as the revealing transaction has been made, he has the option to take measures not to include the transaction in the block with the block number which would result in an unfavorable pseudorandom number outcome for him.

4.2.3 Collaborative Scheme

In this method a number of parties collaborate in order to come up with a random number. RANDAO is probably the most prominent example of this approach [13]. Contracts that require a pseudorandom number can make a request to a generator contract. A fee has to be paid to the generator contract. The creation of the random number follows three main steps. First, each party that wishes to participate in the generation of the pseudorandom number has to provide a sealed seed `hash(seed)` together with a deposit to the generator contract. These sealed seeds have to be provided within a time frame specified by the generator contract. Secondly, each participant has to provide his `seed` to the generator contract. The contract now validates the correctness of each provided `seed` by running it against the `hash()` function and comparing it to the previously provided `hash(seed)`. Seeds that turned out to be valid are then stored in the contract. If a participant does not reveal his seed, his deposit will be confiscated. The deposits of the other participants will be returned and the requesting contract has to wait for a new round where a new pseudorandom number is being generated. Thirdly, the generator contract now generates a pseudorandom number from the collected seeds. Here a simple XOR-function can be used like this `newPseudorandomNumber = xor(seed1, seed2, ... , seedn)`. The variable `newPseudoRandomNumber` is then stored in the generator contract and sent to all the contracts that requested the pseudorandom number in the first place. Now, as an incentive, a part of the profit from the fees paid by the requesting contracts is being sent to the seed providers together with their deposit. Reward settings depend on the implementation and configuration of the generation contract. Here it should be noted that if an XOR-function is used, committing should only be allowed with an unique `hash(seed)`. Otherwise, due to the nature of the XOR-function, committing a second time to the same number essentially removes both `seed` from the equation. This could be abused by an attacker [14]. The delay can generally be expected to be around seven blocks. This is because it can be assumed that nobody can produce seven consecutive blocks and therefore influence the outcome of the pseudorandom number with his mining power. The costs for requesting a pseudorandom number from such a generator contract are, again, dependant on its configuration. The context in which a pseudorandom number is being used should be considered. The more there is at stake from the requested pseudorandom number, the higher the deposits of the seed providers should be. For example, if the pseudorandom number is being used for a bet where the better pays 10 Ether and has a fifty-fifty chance on doubling, the deposit for the seed providers should be higher than 10 Ether. For example, lets assume a 20 Ether deposit. This way, if the better decides to participate in the generator contract as a seed provider he can decide to withhold the revelation of his seed in the second step. By doing so he can get a second chance if he realizes the outcome of the generated pseudorandom number will be to his disadvantage. However, he will still lose his deposit, resulting in a loss of 20 Ether plus a potential gain of 10 Ether. There is yet another way of preventing seeds from being revealed and therefore influence the outcome of the generated pseudorandom number. The attacker does not necessarily have to produce seven consecutive blocks, but he can try to outbid an honest seed provider by filling each of the blocks with transactions that have a greater gas prices than the transaction that would reveal the seed. This prevents the honest seed provider effectively from revealing his seed and therefore the pseudorandom number is not being provided to the contracts that requested it in the first place. They have to wait for the next pseudorandom number that is being provided by the generator contract.

4.2.4 Chainlink

Chainlink is a decentralized oracle network that provides smart contracts with highly available connections to data feeds. This way a smart contract can obtain a variety of real world validated data, like currency prices, sports results or IoT sensor readings. Chainlink does this through an oracle network. Each oracle in the network gets its data from a different source. This solves the problem of relying on a single oracle as an interface to real world data, which would defy the purpose of a decentralized application. Chainlink VRF provides verifiable randomness through its oracle network. As of writing this, Chainlink VRF has served over one million requests for verifiable pseudorandomness to

Binance Smart Chain, Polygon and Ethereum. Chainlink identifies six areas for typical use cases where their service may be of use (gaming and the metaverse, NFT creation and distribution, lucky draws and DeFi, marketing campaigns and loyalty rewards, fair selection and ordering processes, and authentication and security) [4].

4.2.4.1 Accessing Pseudorandomness Through Chainlink VRF

Chainlink VRF (Verifiable Random Function) is a provably-fair and verifiable source of randomness designed for smart contracts [10]. They provide access to pseudorandomness without compromising on security or usability. Other than data feeds the provided randomness cannot be reference data. If the result of randomness was stored on chain, any actor could see and predict the outcome. In order to obtain a random number, a smart contract can make a request for a random number to the Chainlink network. The Chainlink network then responds by aggregating responses from a required number of network participants (Chainlink nodes). Once threshold for required number of participants is met, the result is broadcasted as a single transaction to the requesting smart contract. For each request for a new pseudorandom number, the oracle network first provides a proof that the number has been created. This proof is then being verified and made accessible on the blockchain before the the pseudorandomness can be consumed. This is called a Request and Receive cycle. This ensures that the pseudorandomness cannot be manipulated by miners or by the oracles themselves. The provided pseudorandomness comes at different costs depending on the required security (number of Chainlink nodes involved). Oracles need to be paid in LINK. This demands that the smart contract is in possession of at least the required amount of LINK to request a pseudorandom number from the Chainlink network. LINK conforms to the ERC-677 token standard, an extension of ERC-20. This standard is what enables data to be encoded in token transfers. The payment in LINK happens during the request to the Chainlink network. This is integral to the Request and Receive cycle. Since the oracle and the block miner both have some influence on when the VRF responses appear on the blockchain, the requesting smart contract has to make sure that the order in which the requested pseudorandomness appears on the blockchain cannot be used to manipulate the behaviour of the smart contract. The blockhash of the block which contains the `requestRandomness` call is mixed into the seed of the VRF. Because of this, a powerful miner could in principle fork the blockchain in order to force the `requestRandomness` call to be made in another block resulting in a possibly different outcome for the requested pseudorandomness. The cost for such an attack scales with the amount of blocks the VRF oracle waits until it responds to the request.

4.2.4.2 Chainlink VRF Implementation Details

A smart contract that wants to use the Chainlink VRF service needs to import and extend a contract called `VRFConsumerBase`. It then needs to store two variables, `keyHash` is used to tell the oracle exactly what exactly is being requested (e.g. number of involved oracles), `fee` is the amount of LINK that is to be used in the request. Further mappings between pseudorandomness `requestID` (return from the request) and the user of the pseudorandomness as well as between the actual `result` of the request and the user of the pseudorandomness have to be established. With the `requestRandomness` function provided by `VRFConsumerBase` the `keyHash` and `fee` is being sent to the Chainlink VRF network. The `fulfillRandomness` function is being used by the Chainlink VRF network to provide the requested pseudorandomness to the contract. It returns the `requestId` along with the corresponding pseudorandomness. By overriding the `fullfillRandomness` function the contract is now being able to make use of the pseudorandomness provided by the Chainlink VRF network.

5 The VRF-Approach

As discussed before, the manipulation of provided pseudorandomness is a major security concern for any application that relies on it. An application should try to aim to have its randomness or pseudorandomness be equally uncertain for all participants as well as try to be provably fair. At the same time, the outcome of the randomness should be unpredictable for any adversary. A verifiable Random Function, in short VRF, was proposed by Micali, Rabin, Vadhan [25]. A VRF is set of functions that allows a party to generate a *beta-string* by hashing a *secret key SK* together with a publicly available byte string, the so called *alpha-string*. The generated *beta-string* can serve as pseudorandom number and is unique in the sense that it is hard to replicate without knowledge of the *SK*. At the same time a proof, called the pi-string, is being created which can later be used to verify that the pseudorandom number has in fact been correctly generated from the *SK* and *alpha-string*. In order to do so a *public key PK* that corresponds to the *SK* used to generate the pseudorandom number is required. This allows anyone in possession of the *PK* and the *alpha-string* to verify the soundness of the provided pseudorandom number without revealing any information about the *SK* that was being used. This approach provides a simple way of preventing the provider of the pseudorandomness from manipulating the outcome of the pseudorandom number. A VRF should further have the following properties. Collision resistance, meaning it is hard to find two different inputs that result in the same output. Pseudorandomness, meaning for an observer who does not know the *SK*, the outcome appears to be random. Trusted uniqueness, meaning that for any given *PK*, an *alpha-string* input to the VRF will result in a unique *beta-string*. This is important, if using the same *PK* and *alpha-string* could result in different beta-strings the owner of the *SK* could choose to publish a *beta-string* that is favorable for him. The provided pseudorandom number can be available within one block the soonest. The cost of providing pseudorandom numbers in this manner relies heavily on the individual implementation. Of course the specific usage of a VRF in a contract has to be implemented with sound logic. The internet-draft from Hong Kong University of Science and Technology titled Verifiable Random Functions (VRF) provides blueprints for the implementation of an elliptic curve based VRF as well as for a RSA based VRF [21]. Providing pseudorandomness on Ethereum between two parties using a VRF function requires a total of three interactions between these parties. Assuming party *A* provides and operates the VRF, party *B* has to first commit to an *alpha-string*, then reveal said *alpha-string*, and finally party *A* has to forge the pseudorandomness along with a proof from its own *SK* and the received *alpha-string*. Compared to a commit-reveal-scheme, where both parties commit to a bit-string which later gets revealed and hashed, this approach requires less interaction between parties.

5.1 Components of a Verifiable Random Function

Depending on the definition, a VRF consists of three or four main functions. The first function assigns a *PK* to a corresponding *SK*. The second function consists of two parts, where the first part provides a pseudorandom number and the second part provides a corresponding proof. The third function is a verification function which allows makes use of the afore created proof to verify the soundness of the created pseudorandom number. The second function can in fact be reduced to its second part, which produces only the proof. If implemented this way, a pseudorandom number can simply be derived by calculating a hash from the created proof.

5.1.1 Creating a Key-Pair

Ideally a key-pair is obtained by providing a random seed to a generator function, which then provides a key-pair from a sufficiently large pool. Humans are notoriously bad at coming up with randomness, because of that it is a good idea to use the aid of such a function in order to create the *SK*. The key-pair *(SK,PK)* has to be created with regard to the rest of the VRF. The *SK* is to be kept secret by its holder at all times. Once the *SK* has been leaked, the pseudorandomness property of the VRF is no longer guaranteed, since anyone with knowledge about *SK* will be able to predict the outcome of the VRF.

5.1.2 Creating the Proof

For the prove-function two input parameters are required. The *SK* and the *alpha-string*. These two parameters should be provided from two separate parties and never be used more than once. Otherwise the outcome can be predicted, since each two inputs always produce a unique output. The output comes in the form of a byte-string, called the pi-string. The pi-string later serves as proof of the correct execution of the proof-function. Further it will be used as an input parameter when the pseudorandomness will be created later on.

5.1.3 Pseudorandomness From the Proof

With the proof-to-hash-function the actual pseudorandomness is being generated. This function can be executed by any participants in possession of the pi-string. As input parameter it takes only a single parameter, that is the pi-string created by the proof-function. The produced output is called *beta-string* and can serve as pseudorandomness for the participants involved. The validity of this *beta-string* has to be checked with the verification-function in order to make sure it has not been tampered with.

5.1.4 Verification of the Proof

With the verification-function the integrity of the provided pseudorandomness can be validated. The verification-functions requires three parameter as an input. These parameters are the following. The *PK* corresponding to the *SK* used to generate the proof. The *alpha-string* used to generate the proof. The pi-string that has been created from the *SK* and the *alpha-string*. This function can be executed by any participants in possession of said parameters. The output of the verification-function comes in the form of a boolean and takes the value of either true, in case of correctly formed proof, or false in case of a malformed proof and therefore pseudorandomness that should not be trusted.

5.1.4.1 VRF-Solidity

An implementation for verifying elliptic curve based proofs in Solidity has been provided by Mario Cao [2]. This VRF-solidity library provides two main functions. The `verify(...)` function provides a way to verify provided proofs. This function entails heavy elliptic curve operations and is quite costly at around 2000k of gas. A `fastVerify(...)` function provides yet another way to verify provided proof although at reduced gas costs by using the precompiled `ecrecover` function. `fastVerify(...)` is supposedly requiring only one tenth of the gas costs of `verify(...)` resulting in approximate cost of 200k gas. This however, is being achieved with a trade off in security. Further the library also provides the following three convenience functions. First, the `decodeProof(...)` functions decodes from bytes to a VRF proof. Second, the `decodePoint(...)` function decodes from bytes to a point on the elliptic curve. Finally, the `computeFastVerifyParams(...)` function computes the parameters, being the elliptic curve points, required for the `fastVerify(...)` function. It should be noted that the library does not provide a way of creating proofs. The *SK* should, for obvious reasons, never be used in a transaction over the network. In this regard, it makes sense that the library does not provide a way to create proofs since the *SK* is a necessary parameter that has to be used when creating a proof.

5.2 The Implementation

The following has been implemented. The `CoinFlipper` contract written in Solidity which allows two parties to simulate a fair coin flip. This contract is mainly used as an example of an Ethereum application which requires some sort of randomness. This contract allows both parties to bet money, the winner takes double. The `ECProofFactory`, a contract written in Solidity which mainly allows the formation of elliptic curve based proofs. It also provides the possibility to derive a corresponding *PK* pair from a *SK*. In order to check the validity of a given proof it relies on the afore mentioned vrf-solidity library. The `RSAProofFactory`, a contract written in Solidity which allows the formation of RSA based proofs. From these proofs a pseudorandom byte string, or *beta-string*, can be derived. Finally, it implements a method which allows to verify the proofs created in this manner. It is of utmost importance that none of the functions that use a *SK* as a parameter are being used on chain, since transactions are visible to the entire network and knowledge about the *SK* should remain with the owner. As a workaround a local test chain environment should be used. Alternatively an implementation in the Go programming language allows the formation of RSA- and elliptic curve based proofs. This enables a more convenient way of creating proofs without having to rely on a local test network.

5.2.1 The Coin Flipper

`CoinFlipper` is an Ethereum contract [16] which allows two parties to simulate a fair coin flip. The owner of the CoinFlipper contract accepts a bet. A bet can be made by an Externally Owned Contract (EOA) and entails an amount of Ether [16] which is within the limits specified by the CoinFlipper contract. Once a bet is set, the owner of the CoinFlipper contract has to resolve the bet, and eventually reward the better.

For convenience sake the owner of the CoinFlipper contract will be called the Owner while the better will be called the Better.

Before deployment of the CoinFlipper, the Owner should define the lower limit of accepted bets and the reward factor for a winning bet. Defining the lower limit prevents the Owner from losing Ether on very small bets, since the Owner has to pay gas [16] in order to resolve an ongoing bet. The default reward factor for a bet is set to two, which means if the Better bets 1 Ether he will be rewarded by an amount of 2 Ether if he wins the coin flip. Since the simulated coin flip is fair and the Owner has to invest Gas in order to resolve bets, this will result in the Owner losing Ether should he decide to continuously resolve bets. In order to incentivize the Owner to continuously accept and resolve bets, the reward factor for bet should be adjusted so it is smaller than two and still makes it profitable for the Owner to run the contract even if only bets equal to the lower limit are being played. Obviously, the closer the reward factor is to two, the more attractive it becomes for the Better to bet for the next coin flip. The further the reward factor gets below two, the more attractive it becomes for the Owner to continue resolving bets.

A Verifiable Random Function (VRF) [21] is being used in order to provide a fair coin flip.

At first the Owner commits to a *SK* and provides a paired *PK* which will be made visible in the CoinFlipper contract. Further, the Owner also has to transfer some funds to the CoinFlipper contract, in order to reward winning bets.

As soon as this is done, the CoinFlipper contract will accept valid bets from a Better.

Now bets can be placed. For a bet to be considered valid, it has to suffice a number of predefined criteria. Namely the amount of Ether sent in the bet has to be lower or equal to the predefined upper limit for bets and equal or greater than the predefined lower limit for bets. Further, it must not surpass the current balance of the CoinFlipper contract, otherwise there are not enough funds to reward a winning bet. Additionally there can only ever be one bet at a time. Meaning if currently a bet is ongoing, the CoinFlipper contract will not accept new bets. This limitation has been made for simplicity's sake and could be eliminated if need arises. The Better must also commit to a byte-string by providing the keccak-256 hash value of said byte-string. After the commit lies sufficiently deep within the blockchain (after 7 blocks in the current implementation, which equals to around two minute time difference), the Better has to reveal his byte-string. This extra commit-reveal step is necessary in order to prevent the Owner from censoring bets (taking measures so they are not included in the blockchain). The revealed byte-string is then extended with an integer (uint256) which indicates the number of the current bet, this concatenation forms the *alpha-string*. This prevents that the same *alpha-string* is being used more than once. This way neither the Owner nor the Better will know the outcome of the coin flip in advance. This necessary difference in block height should be adjusted according to the bet size the contract accepts. This measure is necessary in order to prevent the Owner from behaving maliciously. The greater the difference in block height, the harder it becomes for the Owner to behave maliciously. If there is no such difference in block height between the block which contains the bet placement transaction and the block which contains the transaction which resolves the bet, the Owner could collaborate with the miner in order to gain an advantage. The miner could decide to not release a block which would result in a loss for the Owner, or he could simply decide to not to include the transaction which places the bet in the block.

Once the Better has placed his bet, the Owner cannot withdraw funds from the

CoinFlipper contract until he has resolved the bet. Without this mechanism in place, the owner could simply withdraw all the funds from the `CoinFlipper` once he realizes he is about to lose on a bet.

Once a bet has been placed and revealed, the next step is for the Owner to resolve the bet.

Once the bet is revealed, the Owner holds both *SK* and *alpha*. With knowledge of these two parameters the outcome of the VRF becomes completely predictable for the Owner. Because of this circumstance it is of utmost importance that the transaction committing to the bet and the transaction resolving the bet are placed in blocks that have a reasonably big difference in block height.

Resolving a bet can be accomplished by providing a valid proof [21]. A valid proof can be formed with the current *alpha-string*, the *SK* and the use of the `ProofFactory` contract which is discussed later on. Alternatively the Go implementation can be used to form valid proofs. The Go implementation for forming proofs is discussed in section 5.2.4. This proof simultaneously serves as the pseudorandom [28] output of the coin flip. If a valid proof is being provided by the Owner, the better will be rewarded according to the bet reward factor and the amount of his bet if he wins or not otherwise. Now the CoinFlipper contract is open again for new bets.

The more bets played on a CoinFlipper contract, the more trustworthy it becomes in general. A high number of played bets indicates the Owner of the contract is resolving bets with a high likelihood. This incentivises Betters to play on said CoinFlipper contract, which in turn also incentivises the Owner to keep resolving bets.

A different solution to incentivize the Owner to resolve bets would be to implement a required time limit in which each bet has to be resolved. If said time limit passes without the Owner resolving the bet, he would automatically lose and the Better would win. Such a time limit would also prevent the *CoinFlipper* contract from locking up. In the current implementation this can happen if the Owner decides to not resolve a running bet, resulting in both the Owners and the Betters funds becoming stuck within the contract. This is a result of the necessary mechanisms in place that prevent either party from bailing out of a running bet. However, such a time window would also open up the possibility for censorship attacks against the Owner in which the Better could prevent the Owner from resolving the bet within the necessary time window. Resulting in a win for the Better.

5.2.1.1 Possible Attacks Against the Owner

During the resolving phase the Owner of the CoinFlipper has to broadcast his transaction to resolve the bet to the network. This opens up the possibility for a censorship attack against the Owner if the resolving transaction has to be made within a given time window. In case that the Owner wins the bet he still needs to resolve it in order to not automatically lose after the time window in which the bet has to be resolved passes. Once the transaction to resolve the bet is broadcasted to the network the Better will realize that he has lost the bet. He can now take steps to prevent the bet from being resolved. This could be done by doing transactions which are willing to pay high gas fees. This approach is only viable for very high value bets that would be so heavy that they could justify the required gas costs for such an attack. Alternatively the Better could simply collaborate with miners, in order to exclude transactions from the current block which are trying to resolve the bet.

A possible solution would be to implement another commit-reveal part to the resolving phase. Here the Owner would commit to the resolved outcome by publishing `hash(result)`. After a given time frame, he would then publish the `result` as well. If the time frame is large enough, this approach protects the Owner from falling victim to a censorship attack as described above. However this would make the whole process slower and now the communication requirements are at the same height as with a pure commit-reveal scheme.

5.2.1.2 Censorship by Miner

In any case where a party has to make a transaction to the blockchain in order to not lose a previously stacked amount of Ether, the party becomes vulnerable to a censorship attack. In the CoinFlipper contract this is the case once a party has published the hash of a later to be revealed seed. Here the assumption is being made that a miner does not possess the necessary mining power to censor a transaction throughout several consecutive blocks (7 blocks in the current implementation).

5.2.2 The Elliptic-Curve Proof Factory

The `ProofFactory` contract covers three use cases. First, it allows to derive a *PK* pair to a given *SK*. Second, given a *SK* and an alpha byte string it can create valid proofs for an Elliptic Curve VRF according to the description in the internet-draft from Kong University of Science and Technology titled Verifiable Random Functions (VRF) [21]. Thirdly, it can validate proofs with help of the vrf-solidity library [2].

Some of the functions of this contract require a *SK* as a parameter, because of that these functions should never be used on chain. A solution would be to use an isolated private node on which the contract is then run. Another solution would be an implementation of the same functionality in another programming language. This could make it more convenient to run locally.

In order to derive a *PK* from a *SK*, the function `derivePublicKey()` makes use of the `EllipticCurve` library [19]. Here an elliptic curve multiplication is being performed with the *SK* as the scalar and the generator point of the elliptic curve.

The function `createProof` uses the *SK* and the alpha byte string as parameters in order to generate a proof, also called Pi-String. This is achieved through the steps laid out in the internet-draft Verifiable Random Functions (VRF) [21]. The algorithm works as follows:

Algorithm 1: ECVRF_prove

```
Data: SK, alpha_string
Result: pi_string
PK = derivePoint(SK, G);
hPoint = hashToTryAndIncrement(PK, alpha_string);
hString = encodePoint(hPoint);
gammaPoint = derivePoint(SK, hPoint);
k = nonceGeneration(SK, hString);
kB = derivePoint(k, G);
kH = derivePoint(k, hPoint);
c = hashPoints(hPoint, gammaPoint, kB, kH);
s = (k + c * SK) mod PP;
pi_string = concatenate(encodePoint(gammaPoint), intToString16(c),
  intToString32(s));
```

Step 1: The *PK* has to be derived from the *SK* via elliptic curve multiplication with the *generator point G* of the curve.

Step 2: *hPoint* is a point on the elliptic curve. It can be derived by hashing the concatenation of the *PK* and the *alpha_string* repeatedly (together with an added counter) until a point is found which lays on the curve.

Step 3: In *hString* the found Point from step 2 is simply put into a string representation in which the y-coordinate is represented with an added prefix.

Step 4: The *gammaPoint* has to be derived from the *SK* via elliptic curve multiplication with the in step 2 generated *hPoint*.

Step 5: Derives a nonce pseudo randomly and uniformly from *SK* and *hString* generated in step 3. This prevents an attacker from deriving the *SK* after observing multiple proofs generated from the same *SK*.

Step 6: An elliptic curve multiplication between the scalar *k* and the generator point *G* as well as *hPoint* is performed. The resulting points *kB, kH* on the curve are then hashed (sha256) together with *hPoint* and *gammaPoint*. The first half (most significant bits) of this hash value is then assigned to *c*.

Step 7: The variable *c* is multiplied by *SK* added to the nonce *k* modulus the prime order of the used curve.

Step 8: Here the *pi_string* is formed by concatenating the encoded *gammaPoint*, part of *c* converted to an integer and part of *s* converted to an integer.

Step 9: The *pi_string* is being returned.

The proof, or Pi-String, itself also serves as the source for generated pseudorandomness. The vrf-solidity library [20] provides a function called `decodeProof` which decodes a valid proof into its single components. With the function `gammaToHash`, also found in the vrf-solidity library [20], said components can be used to derive a pseudorandom bytestring.

Once a proof has been created it can be validated with the help of the vrf-solidity library [20]. This process requires the use of the *PK*, the proof itself and the alpha byte string.

5.2.3 The RSA Proof Factory

The ProofFactory contract also covers three use cases. First, given a *SK* and an alpha byte string it can create valid proofs for a RSA Full Domain Hash VRF according to the internet-draft Verifiable Random Functions [21]. Second, from the proof it can create a pseudorandom byte string. Thirdly, it can validate said proofs.

Some of the functions of this contract require a *SK* as a parameter, because of that the contract should never be used on chain. A solution would be to use an isolated private node on which the contract is then run. Another solution would be an implementation of the same functionality in another programming language. This could make it more convenient to run locally.

The function *createProof* uses the *SK* and the alpha byte string as parameters in order to generate a proof, also called Pi-String. This is achieved through the steps laid out in the internet-draft Verifiable Random Functions (VRF) [21].

Algorithm 2: createProof

```
Data: SK, alphaString
Result: piString
oneString = I2OSP(1, 1);
EM = MGF1(concatenate(oneString, I2OSP(k, 4), I2OSP(n, k), alphaString),
  k-1);
m = OS2IP(EM);
s = RSASP1(SK, m);
piString = I2OSP(s, k);
```

Step 1: *I2OSP* produces an octet string of a desired length from an integer. The first parameter is the integer which is to be converted into an octet string. The second parameter is the desired length of the octet string. *oneString* receives the converted integer in octet string form.

Step 2: *MGF1* is a mask generation function that takes an octet string of arbitrary length as a seed to generate a pseudorandom octet string of desired length (here *k-1*). *sha256* is used as the underlying hash algorithm.

Step 3: *OS2IP* converts an octet string to a non-negative integer.

Step 4: *RSASP* is a RSA signature primitive. Given *SK* and the in step 3 generated *m*, it raises the *m* to the secret RSA exponent modulo n.

Step 5: *piString* is formed by transforming the number *s* in step 4 into an octet string of length k (length of the RSA modulus in octets).

Step 6: The *piString* is being returned.

The proof, or Pi-String, itself also serves as the source for generated pseudorandomness. The function called `proofToHash` can be used to derive a pseudorandom bytestring from said proof.

Algorithm 3: proofToHash

```
Data: piString
Result: betaString
twoString = I2OSP(2, 1);
betaString = Hash(concatenate(twoString,piString));
```

Step 1: *I2OSP* produces an octet string of a desired length from an integer. The first parameter is the integer which is to be converted into an octet string. The second parameter is the desired length of the octet string. *oneString* receives the converted integer in octet string form.

Step 2: The concatenation of *twoString* and *piString* is hashed (*sha256*) and assigned to *betaString*.

Step 3: *betaString* is returned as the produced pseudorandom octet string.

Once a proof has been created it can be validated with the help of the verify function. This process requires the use of the *PK*, the proof itself and the alpha byte string and will return a boolean with value true, if the proof is valid, or false if the proof is invalid.

Algorithm 4: verifyProof

```
Data: PK, alphaString, piString
Result: boolean
s = OS2IP(piString);
m = RSAVP1(PK, s);
EM1 = I2OSP(m, k-1);
oneString = I2OSP(1,1);
EM2 = MGF1(concatenate(oneString, I2OSP(k, 4), I2OSP(n, k), alphaString),
  k-1);
return EM1 == EM2;
```

Step 1: *OS2IP* converts an octet string to a non-negative integer.

Step 2: RSAVP1 is a RSA verification primitive. Given a *PK* and *s*, it raises the *s* to the public RSA exponent modulo n.

Step 3 & 4: *I2OSP* produces an octet string of a desired length from an integer. The first parameter is the integer which is to be converted into an octet string. The second parameter is the desired length of the octet string. *oneString* receives the converted integer in octet string form.

Step 5: *MGF1* is a mask generation function that takes an octet string of arbitrary length as a seed to generate a pseudorandom octet string of desired length (here *k-1*). *sha256* is used as the underlying hash algorithm.

Step 6: The output is determined by whether *EM1* and *EM2* are equal or not.

5.2.4 Generating Key-Pairs and Proofs Offline with Go

Since the secret key *SK* must be kept secret in order to guarantee the pseudorandomness property of the VRF, it should never be passed as a transaction over the network. Although both contracts, `ECProofFactory` and `RSAProofFactory`, implement a method to create proofs as well as a method to create key pairs, they should never be used on a live network. This is due to the fact that these methods are using the *SK* as a parameter. If once *SK* is part of a transaction, it will be visible to anyone on the network. These methods can be used on a local test-network, if the user is confident enough that no one else will gain access to his *SK*. To provide a more convenient method for creating keypairs and calculating proofs without having to access a local test-network, an implementation of these methods in Go is being provided. Once the proof is being formed, it can be transmitted to the network, since it does not provide any information about the *SK*. The Go implementation renders the solidity-functions which create keypairs and proofs redundant. However for convenience and testing purposes these functions will remain in the code.

5.2.4.1 Creating Proofs for Elliptic Curve Based VRFs with Go

The Go-File `createProofEC.go` contains a method to create key pairs *(SK,PK)* from scratch as well as a method to create a *PK* to a given *SK*. In order to form proofs, the library `crypto/ecdsa` is being used.

5.2.4.2 Creating Proofs for RSA Based VRFs with Go

The Go-file `createProofRSA.go` contains a method to create key pairs *(SK,PK)* from scratch. In order to form proofs, the guidelines in the internet-draft from Verifiable Random Functions (VRF) have been followed [21]. The algorithm works in the same manner as **Algorithm 2** described in section 5.2.3.

6
Conclusion and Future Work

Providing provably fair randomness or pseudorandomness to a blockchain like Ethereum can be an intriguingly difficult endeavor. Relying on an oracle for true randomness defies the underlying principle of a distributed system. As we have seen, using the blockhash (or any other parameter that stems from within the blockchain) as a source is not a good solution due to the vulnerability to numerous attacks that mostly involve some form of collaboration with a miner. Collaborative schemes are vulnerable to censorship attacks and require a large commitment to deposits from the participants. The commit-reveal approach is a relatively easy solution, however it requires the distribution of a new commit for every single use. Using a VRF allows using the same public key over and over again, provided the secret key does not get compromised. When implementing the use of a VRF into a contract various security aspects have to be considered in order to not fall victim of common attacks. The most convenient and convincing solution to provide pseudorandomness to a Ethereum contract is by using Chainlink VRF. Chainlink VRF requests can easily be made for different security requirements at different fees.

As an outlook for future work a comparison in regard to costs and security between the current implementation of the CoinFlipper could be made with a contract that requests its pseudorandomness from Chainlink VRF. Further the current implementation of the CoinFlipper could be extended in the following manner. Allowing multiple games at the same time. Implementing the requirement for a commit to each resolved bet (this would entail an analysis for possible censorship attacks against the Owner). Since the CoinFlipper contract provides a way to make bets between users, it would be interesting to explore the requirements for a slot-machine-like contract that allows bets between a contract and users.

Bibliography

[1] "4 letter words." https://www.wordgamedictionary.com/word-lists/4-letter-words/. Accessed: 2021-09-24.

[2] "Announcing our verifiable random function (vrf) library in solidity." https://medium.com/witnet/announcing-our-verifiable-random-function-vrf-library-in-solidity-c847edf123f7. Accessed: 2021-08-04.

[3] "The brownian movement." https://www.feynmanlectures.caltech.edu/I_41.html. Accessed: 2021-08-17.

[4] "Chainlink vrf hits one million requests." https://www.bsc.news/post/chainlink-verifiable-random-function-hits-landmark-1-million-requests. Accessed: 2021-10-11.

[5] "Cryptographic random numbers." https://theworld.com/~cme/P1363/ranno.html. Accessed: 2021-08-04.

[6] "Cryptokitties." https://www.cryptokitties.co/. Accessed: 2021-09-11.

[7] "Diceware for passphrase generation and other cryptographic applications." https://theworld.com/~reinhold/diceware.txt. Accessed: 2021-08-17.

[8] "The economic impact of random rewards in blockchain video games." https://blog.chain.link/the-economic-impact-of-random-rewards-in-blockchain-video-games/. Accessed: 2021-09-11.

[9] "How not to run a blockchain lottery." https://hackingdistributed.com/2017/12/24/how-not-to-run-a-blockchain-lottery/. Accessed: 2021-08-04.

[10] "Introduction to chainlink vrf." https://docs.chain.link/docs/chainlink-vrf/. Accessed: 2021-08-04.

[11] "Primer on elliptic curve cryptography." `https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/`. Accessed: 2021-09-24.

[12] "Quadratic sieve." `https://mathworld.wolfram.com/QuadraticSieve.html`. Accessed: 2021-09-24.

[13] "Randao: Blockchain based verifiable random number generator." `https://www.randao.org/`. Accessed: 2021-01-09.

[14] "Rigging randao: Defeating the crypto world's favorite "trustless" random number generator." `https://revelry.co/resources/development/critical-randao-vulnerability/`. Accessed: 2021-10-11.

[15] "Solidity." `https://docs.soliditylang.org/en/v0.8.7/`. Accessed: 2021-09-12.

[16] A. Antonopoulos and G. D, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018.

[17] D. R. L. Brown, "Standards for efficient cryptography (sec 2): Recommended elliptic curve domain parameters." `http://www.secg.org/sec2-v2.pdf`, 2010. Accessed: 2021-10-11.

[18] C. S. Calude, "Quantum randomness: From practice to theory and back," in *The Incomputable: Journeys Beyond the Turing Barrier* (S. B. Cooper and M. I. Soskova, eds.), Theory and Applications of Computability, pp. 169–181, Springer International Publishing, 2017.

[19] M. Cao, "elliptic-curve-solidity." `https://github.com/witnet/elliptic-curve-solidity`. Accessed: 2021-10-03.

[20] M. Cao, "vrf-solidity." `https://github.com/witnet/vrf-solidity`. Accessed: 2021-10-03.

[21] S. Goldberg, L. Reyzin, D. Papadopoulos, and J. Včelák, "Verifiable Random Functions (VRFs)," Internet-Draft draft-irtf-cfrg-vrf-10, Internet Engineering Task Force, Nov. 2021. Work in Progress.

[22] R. L. Graham, "Euclidean ramsey theory," in *Handbook of Discrete and Computational Geometry, Second Edition* (J. E. Goodman and J. O'Rourke, eds.), pp. 239–254, Chapman and Hall/CRC, 2004.

[23] M. Haahr, "RANDOM.ORG: true random number service." `https://www.random.org`.

[24] D. E. E. III, S. D. Crocker, and J. I. Schiller, “Randomness recommendations for security,” *RFC*, vol. 1750, pp. 1–30, 1994.

[25] S. Micali, M. Rabin, and S. Vadhan, “Verifiable random functions,” in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pp. 120–130, 1999.

[26] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[27] T. Shinbrot, C. Grebogi, J. Wisdom, and J. Yorke, “Chaos in a double pendulum,” *American Journal of Physics*, vol. 60, pp. 491–499, June 1992.

[28] S. P. Vadhan, “Pseudorandomness,” *Found. Trends Theor. Comput. Sci.*, vol. 7, no. 1-3, pp. 1–336, 2012.

[29] J. Walker, “HotBits: genuine random numbers, generated by radioactive decay.” `https://www.fourmilab.ch/hotbits/`.