Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Viktor Frede Andersen

Distributed lottery on Ethereum

Implementation of a tournament based distributed lottery on Ethereum

Master's thesis in Computer Science
Supervisor: Letizia Jaccheri
June 2019

NTNU
Norwegian University of Science and Technology

Viktor Frede Andersen

Distributed lottery on Ethereum

Implementation of a tournament based distributed lottery on Ethereum

Master's thesis in Computer Science
Supervisor: Letizia Jaccheri
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Preface

This is a Master's thesis carried out during the spring semester of 2019 for the completion of the 5-year computer science program at Norwegian University of Science and Technology, Trondheim. The idea for a tournament based lottery on Ethereum was inspired by the works of Miller and Bentov and Bartoletti and Zunino. The search for a thesis went over a long time during which I studied blockchain applications and learned about the blockchain space.

The readers of this report are assumed to have a background in computer science with some knowledge of distributed systems and blockchains.

11-06-2019

Viktor Frede Andersen

Acknowledgment

I would like to thank the following persons for their great help during the thesis work:
Mariusz Nowostawski for being an excellent supervisor and mentor with patience and understanding.
My mom Liv Unni Andersen and dad Jan Kjetil Andersen for mental support, guidance, and for pushing me when progress was slow.
All contributors to the academic work which I built this thesis on, including open source contributors and advocates of Bitcoin, Ethereum, and associated software projects.

V.F.A.

Abstract

Distributed lotteries on the internet is an interesting application with uses in gambling and other areas. Blockchains that function as smart contract platforms can be used to both transfer value and enforce protocols for multiparty computing without relying on a trusted intermediary. This has made it possible to design lotteries with verifiable randomness and with a guarantee of successful completion.

One promising design of a distributed lottery on a blockchain is based on a tournament of digital coin tosses. This thesis explores the feasibility of such a lottery through making an implementation and doing measurements. The implementation is made for the Ethereum blockchain, which is currently the leading platform for applications using smart contracts. The lottery is assessed by measuring transaction costs and transaction demand, as well as by discussing the security of the lottery in the context of known security issues for blockchain applications and web applications.

We successfully implement a lottery prototype which is likely to work for up to about 100000 participants. Several directions of further research to improve the scalability are identified and discussed. We find that the most concerning security issue is transaction censorship by a powerful collusion of opportunistic miners, which might be an issue for lotteries with a very large prize.

Abbreviations

P2P = Peer-to-Peer
PoW = Proof-of-Work
VRF = Verifiable Random Function
PRNG = Pseudo-Random Number Generator
API = Application Programming Interface
MPC = MultiParty Computation
BVM = Bitcoin Virtual Machine
EVM = Ethereum Virtual Machine
CA = Contract Account
EOA = Externally Owned Account
dApp = Decentralized Application
RPC = Remote Procedure Call
DoS = Denial-of-Service
PoS = Proof-of-Stake
HDS = Hierarchical Deterministic Secrets

Contents

List of Figures

List of Tables

Listings

1 Introduction

A lottery can be defined as a random distribution of a prize fund to a set of participants, or more generally, a random selection of a subset of winners from a larger set of participants to receive some privilege. A lottery is often associated with its use in gambling where one becomes a participant by contributing to a prize fund by buying a ticket, whereupon a set of winning tickets is randomly drawn and a part of the fund is redeemable by owning a winning ticket. Lotteries have also been used in important societal functions such as leader election in democratic governance [1] or proof-of-stake systems, drafting of soldiers to war [2], jury selection, and distribution of scarce goods [3], or as a game used in fundraising to a charitable cause.

Since the stakes in a lottery can be quite high, as with large cash prizes or being drafted to a war, it is crucial that the result is unpredictable and unbiased. For the lottery to achieve its purpose of ultimately distributing the prize, it is important that there is consensus of the result, and that the rules are enforced. Achieving unbiased randomness and consensus is typically done in one of two ways: 1) auditing and public verification of every step of the protocol, i.e. paper tickets being sold and blindly drawn from a basket, as is common for small-scale charity lotteries, or 2) trust in a central authority to conduct the lottery fairly and according to the rules, which is typical for national lottos and government-backed lotteries. In lotteries of very large scale, only the second way has typically been feasible.

With the advent of the internet and peer-to-peer (P2P) online communications, it is natural to explore the possibility of porting lotteries to this domain. A lottery is a system of several components and is conducted in a process where a set of participants is defined, and a subset of winners is randomly chosen. Components of the system include a random process to decide the winners, an authentication process to decide the participants, a mechanism to distribute the prize, and often a way to handle payments. A computerized lottery that is accessed over the internet can be made by having an authoritative server handle the entire lottery process. While this approach is not trivial and requires careful design [4, 5, 6, 7, 8, 9], it assumes that the authoritative server or an auditor can be trusted to make sure the process is conducted correctly. Lotteries where we cannot assume trustworthy actors or a trusted intermediary will be referred to as *distributed lotteries*, and is the type of lotteries this thesis is concerned about.

In a distributed lottery, all the components of the lottery must work in a distributed network setting. It should be resistant to adversarial behaviour by participants such as sybil attacks [10] or attempts to manipulate the random process, as well as a dishonest organizer. While there exists provably fair protocols to play games that involve randomness between non-trusting players [11, 12, 13, 14], the scale of a lottery makes these protocols impractical. The scaling problem can be solved by making a random seed unpredictable with delay a function [15], or by delegating the random process to a semi-trusted committee [16]. Even with good approaches to the random process, other components of a lottery, such as handling payments and enforcing the organizer to respect the result of the random process, can be challenging to handle in a distributed setting.

Blockchain platforms with cryptocurrencies mark an important shift in P2P computing, as they make it possible to transfer value and arrive at global consensus without a trusted intermediary [17]. Due to the scripting capabilities of several blockchain platforms, they support implementations of various financial instruments and games involving money. There has been gambling and lottery applications on Bitcoin and Ethereum for some time, and the topic of lotteries has been discussed in the academic literature [18, 19, 20, 21, 22, 23]. One class of lotteries uses digital coin tosses organized in a tournament to fairly select a winner. This scheme can in theory support a large amount of participants while also maintaining a high degree of resistance from collusion as well as verifiability of the entire lottery process.

A lottery of this sort has as far as we know been outlined as a proof-of-concept [22], but not been analyzed and discussed in detail. Applications of smart contracts have to cross a gap from theoretical soundness to practical feasibility. By analyzing an implementation in detail, measuring the transaction costs to deploy and play it, and discussing its security and other issues, we hope to get a better understanding of lotteries on a blockchain.

1.1 Thesis statement

Miller and Bentov in [22] outline a fair lottery that can be implemented both on the Bitcoin platform and on other blockchains with a more expressive scripting language such as Ethereum. The authors note that an implementation of their lottery on Ethereum is significantly less complex and more scalable than the same scheme on Bitcoin. For this reason, as well as the fact that Ethereum has a rich ecosystem of developer tools and user interfaces that make smart contract applications accessible, we decided to implement a full version of the lottery on Ethereum.

Although there exists several lotteries on Ethereum already, to our knowledge none are able to achieve fairness and resistance to manipulation to the degree of

Miller and Bentov's lottery. The purpose of creating an implementation is to get some insight into the viability of the lottery in a practical sense. While we will discuss theoretical considerations for the lottery, we expect to discover new considerations related to security, usability, and performance with a working implementation.

The theme of this thesis will be to assess the feasibility of a lottery similar to that in [22] through making a full implementation. It will have an exploratory aspect in that we expect to discover limitations and constraints for the lottery in a real setting, and a more practical aspect in that a working proof-of-concept application will be made.

1.2 Methodology

1.2.1 Literature review

A systematic literature review on distributed lotteries was conducted prior to writing this thesis in order to place the work in the context of published literature. Sources were collected by conducting a keyword search in online databases for published academic papers. The results from the search were restricted to the fields of computer and information science, as there were many irrelevant hits from different fields. All papers that included a design of a distributed lottery were included in the review. Additional relevant works were found in the reference list of papers found through the keyword search.

Hits from the keyword search were first filtered based on their title and abstract. Papers that were clearly about a different topic than distributed lotteries were not included. Another round of filtering was done by reading the introduction, table of contents, and conclusion where papers that did not include a lottery design or something very similar were discarded.

Both the Scopus [1] and Web of Science [2] databases were queried, but the relevant hits from Web of Science were a strict subset of the relevant hits from Scopus.

Table 1: Keyword search on Scopus.

term	hits	filtered hits	lottery designs
Term 1	13	12	7
Term 2	44	20	14

Table of results from literature search. See terms below.

- Term 1: (bitcoin OR blockchain) AND (lottery OR lotteries)

[1] Scopus https://www.scopus.com/
[2] Web of Science v5.32 https://apps.webofknowledge.com/

- Term 2: (verifiable OR verifiability OR p2p OR "peer-to-peer" OR distributed) AND (lottery or lotteries)

Table 2: Designs of lottery schemes.

from keyword search	from references	total
19	6	25

1.2.2 Data collection

A working implementation of a distributed lottery was implemented on a local version of the Ethereum blockchain. This is a fully working proof-of-concept application that can be simulated and interacted with in order to collect data and get a sense of what a live implementation would look like. We collected data of transaction costs necessary to set up a lottery of various sizes by measuring the gas usage of simulations. We found the number of interactions a participant in the lottery is needed to perform with the blockchain in order to complete the entire lottery process successfully. Data for gas price, ether price, and transaction throughput on the Ethereum blockchain was gathered from Etherscan [3] and ETH Gas Station [4]. Additional data from previous published papers are used when discussing results, and these will be clearly referenced.

1.2.3 Data analysis

The data collected from simulations were of high quality and did not need preprocessing. The data will be presented in Chapter 4, and is used as the basis for analyzing certain properties of our lottery implementation.

1.3 Thesis structure

Chapter 2 introduces the reader to background material relevant to the topics discussed in the thesis.
Chapter 3 presents the implementation and design choices of the distributed lottery on Ethereum this thesis investigates.
Chapter 4 is about the results generated from experiments with the implementation and an analysis of the viability and security of the implementation.
Chapter 5 is a discussion on the the work and process that was used when doing the work for this thesis.
Chapter 6 summarizes the thesis, contains the conclusion, and presents ideas for future work.

[3] Etherscan https://etherscan.io/
[4] ETH Gas Station https://ethgasstation.info/

2 Background

2.1 Cryptography

2.1.1 Secure hash functions

Hash functions are deterministic functions that take input of arbitrary finite length, a *preimage*, and produce an output of fixed length, a *hash* [24, p. 153]. When the length of a preimage is much larger than the output length, the hash function is effectively a lossy compress function. A hash function can be used for various purposes, and can be designed to have properties fit for its purpose. A secure hash function is one that has properties that makes it useful for a range of cryptographic uses.

Properties of secure hash functions

Collision resistance. A hash function is collision resistant if it is infeasible to find two preimages m and m' where $m \neq m'$ so that $Hash(m) = Hash(m')$. Such m and m' must exist, as the domain of arbitrary length inputs is larger than the range of fixed length outputs. But the length of outputs in secure hash functions can be so large that it's computationally infeasible to find such preimages. A hash function that is collision resistant will implicitly also be *preimage resistant* and *second-preimage resistant*. Second-preimage resistance means that given a preimage m it is infeasible to find a $m' \neq m$ so that $Hash(m) = Hash(m')$. Preimage resistance essentially means that a hash function is *one-way*.

One-way. A secure hash function is a one-way function in that it should not be possible to find the preimage m given a hash $h = Hash(m)$ by any other means than guessing all possible m, i.e. a secure hash function should be infeasible to invert. In cryptographic applications, the one-way property makes it possible to publicly share hashes of secret preimages without risk of compromising the secrets. We will see that this can be useful when a secret s at a later point will be revealed, as it makes it possible to verify that a previously shared hash h is indeed the hash of a secret by verifying that $Hash(s) = h$.

Avalanche effect. The avalanche effect is a term used to describe that a small change in a function's input will have a large effect on its output. A strict criterion for the avalanche effect is that any change in the input of a hash function causes each bit in the output to flip with a probability of 0.5. If similar hashes are more likely to come from similar preimages, it would be possible to make good guesses when attempting to reverse a hash function by employing statistical analysis, as

has been demonstrated in works such as [25].

Some applications of secure hash functions

Hashing to verify message integrity and authenticity. Hash functions can be used to verify the integrity of a message [24, p. 158–164]. We assume one party, Alice, has received the hash h of a long message m from a trusted party. Another party Alice does not trust, Bob, claims to have m and offers to transmit it to Alice. Since Alice knows the hash of the message, she can retrieve the message from Bob and independently verify the message's integrity and authenticity by verifying that $Hash(m) = h$. This makes it possible to e.g. retrieve large files from untrusted parties while only retrieving small hashes from trusted parties.

Hashes as digital fingerprints. The hash of any digital representation of information, such as a file, can be considered as practically unique if the hash function is collision resistant. This enables the hash be used as an identifier or fingerprint. This is useful in content addressing and resource lookup in distributed file systems. This concept is also useful for deduplication, as it provides a method for discovering identical files within a storage system [24, p. 182-183].

Data structures linked by hashes. A block of data can be hashed and addressed by its hash, as a data block's hash serve as an identifier. If a data block contains the hash of another block, we can create acyclic data structures with useful properties in integrity and authenticity. This can be used to verify membership of some item in a collection without knowing the entire collection.

A common data structure that uses data linked by hashes is the Merkle tree [26]. A collection of data blocks are ordered in $\{block_0, block_1, ..., block_n\}$. The ordered items form the leaves of a full binary tree. The collection is padded with empty items if the number of items is not a power of two. Each internal node is identified by the hash of the concatenation of its children, and each leaf node by the hash of its content. The root of any subtree will be a dependent on the data of the items in its leaf nodes, and a change of the data in any item of the collection will cause a change in the identifiers of the root of each subtree that contains the item. This makes it possible to verify that an item exists in a collection if the verifier only knows the identifier of the root node. A prover needs to provide the item as well as the identifiers of all nodes in the branch from the root node to the leaf node of the item.

Hash functions as random oracles. A secure hash function is collision resistant. The implication of this is that no observer can know the preimage by seeing only the hash, or feasibly guess the hash of a preimage without calculating it. The avalanche effect makes it so that the mapping of inputs to outputs form a uniform distribution of the range with all inputs being independent. This is a stronger assumption than simply preimage and second preimage resistance, and is not formally proven by

any hash algorithm [24, p. 179-181]. Still, many applications assume that secure hash functions' outputs are uniformly distributed. Since this property is not formally proven, a *random oracle model* assumption is often made for hash functions. Under the random oracle model, a given hash function is assumed to have outputs uniformly distributed and inhibit the avalanche effect.

Hashes in commitment schemes. A party can publish the hash of a message at some time t to prove that it possessed the message at time t. The committing party does not need to reveal anything about the message if we assume the hash function is preimage resistant [24, p. 187–189]. This concept can be used in commitment schemes [27] to create an unpredictable, but reproducible number that can be used as a seed to a pseudo-random function. This allows untrusting parties to agree on random numbers and can be used to implement digital coin tosses [12, 18].

Hashing in proof-of-work (PoW). Under the random oracle model, all values within the range of a hash function are equally likely to be the output for any given input. We can define a subspace within the range of any size and know the probability of finding an input that maps to that subspace. E.g. if the range is all bitstrings with length l, we can define a subspace of the range to be all bitstrings of length l where the first z leading bits are zero. The probability of an input mapping to this subspace will be $p_z = \frac{1}{2^z}$. If we assume that calculating a hash has a cost, we see that finding an input that maps to a limited range has an expected cost. This makes it arbitrarily hard rather than infeasible to find a preimage that maps to a certain subspace. While finding such a preimage may require millions of trials, verifying that an input maps to the subspace requires only one calculation of the hash function.

This fact has been used to enforce a price to be paid in number of expected computations to be executed in some online protocols [28, 29]. In order to prevent reusing of known hashes, a unique challenge c can be issued to the prover who pays the price. The prover must produce a hash so that $Hash(c||n) \in SS$ where SS is the subspace, $||$ is the concatenation operator, and n is a nonce which can be any value.

2.1.2 Digital encryption

Digital encryption is realized by a set of algorithms $Encrypt(m, k) \Rightarrow c$ and $decrypt(c, k) \Rightarrow m$. Here, m is a message, c is a cipher and k is a key. A message is securely encrypted if it is computationally infeasible to decrypt the message without knowing k. In *symmetric cryptography*, the same key is used for encryption and decryption, while in *asymmetric cryptography*, different keys are used for encryption and decryption. The former is also known as private-key encryption and is typically used for secure communication over an insecure channel while the latter is also known

as public-key encryption and used for both secure communication and to enable digital signatures.

Public and private key pairs

In [30], Diffie and Hellman describe an interactive protocol that makes it possible to perform an interactive secure key exchange, often called a handshake, between parties over an insecure channel where they can securely generate a secret key. In addition to what became known as the Diffie-Hellman key exchange, the seminal 1976 paper also introduced the concept asymmetric cryptography, and is often considered to mark the beginning of a cryptographic revolution that made it possible to use very strong cryptography using commodity computing devices and public infrastructure.

In a public-key encryption scheme, a pair of keys (pk, sk) are generated so that one can be used to decrypt messages that are encrypted by the other. The keys in the pair serve different roles; a public key pk is widely disseminated and is used to encrypt messages intended for the receiver who knows the corresponding secret key sk – the other key in the pair, which can decrypt the message [24, p. 370]. This makes it possible for parties to communicate confidentially by only knowing each other's public key in advance, i.e. no secure key exchange in advance is necessary.

A secure network using public-key schemes has some scaling advantages as well. In a network of n nodes where each node wishes to communicate securely and privately with any other node, the network would need to store n public keys and n private keys in aggregate. With a private-key scheme, there would have to be a key for each pair of nodes, i.e. one key for each edge in a fully connected graph of n nodes, which is $(n-1)^2$ keys in aggregate. We see that the former scales linearly and the latter scales polynomially with regards to keys stored.

2.1.3 Digital signatures

Digital signatures make it possible to prove that a message has been signed by a certain party. A digital signature scheme consist of a set of probabilistic polynomial time algorithms: $Generate(n)$ which outputs an asymmetric key pair (pk, sk) where n is a security parameter; $Sign(sk, m)$ which outputs a signature σ; $Verify(pk, m, \sigma)$ which outputs $true$ iff $Sign(sk, m) = \sigma$ or $false$ otherwise [1]. Such a scheme can be implemented by encrypting a message with a private key that is known to belong to a certain party. If we assume a network where participants are aware of each other's public keys, one can authenticate oneself by encrypting a predefined message. Consider the public-private key pair (pk_{Alice}, sk_{Alice}) and the predefined challenge c. A signature σ is created with $\sigma := Encrypt(c, sk_{Alice})$. The signature is verified if $c = Decrypt(\sigma, pk_{Alice})$. The verifier will be able to authenticate Alice

[1] With negligible probability of this not happening.

and Alice will not be able to repudiate that she did sign the challenge. Typically, a challenge is signed together with a nonce such as a timestamp to prevent replay attacks, where the verifier poses as Alice by reusing the signature.

While signing predefined challenges is useful for authentication, the same principle can also be used to sign arbitrary data such as messages generated by the signer. Instead of encrypting the entire message and using that as a signature, typically only the hash of the message is signed and sent along with the message. The sender, Alice, hashes the message m so that we have the hash $h = Hash(m)$. The signature is $\sigma := Encrypt(h, sk_{Alice})$. The receiver will hash the message and verify its integrity and authenticity by verifying that $Hash(m) = Decrypt(\sigma, pk_{Alice})$.

2.2 Verifiable randomness

2.2.1 Introduction to verifiable random functions (VRF)

Generating random numbers is useful in many applications. Random numbers can be generated using pseudo-random number generators (PRNG), which take a seed as input and produce a series of uniformly random and unpredictable bit strings. A limitation of PRNGs is that they are not verifiable. If one is presented a random number generated from a PRNG, one cannot verify that the number is actually random without knowing the seed, as it could be an arbitrary number. If, however, one knows the seed, the random number is no longer unpredictable to the verifier. In a setting where unpredictable random numbers are needed, and we cannot assume a trusted party to generate them, we have to be able to generate a random number which can be verified to have been unpredictable at the time of computation.

Micali et al. in [31] define a VRF as 3 algorithms: a function generator $Gen()$, a function evaluator $(Fun_1(), Fun_2())$, and a function verifier $Verify()$. $Gen()$ receives a unary string, a security parameter, as input and generates an asymmetric key pair (pk, sk). $Fun_1()$ receives sk and an input x, and generates a random value v, while $Fun_2()$ generates a proof $proof$ from sk and x. $Verify()$ takes $(pk, x, v, proof)$ as input and outputs either $true$ iff $proof = Fun_2(sk, x) and v = Fun_1(sk, x)$ or $false$ otherwise.

A set of participants that do not trust each other needs to generate a random number that is unpredictable by all. The random number cannot come from an outside source, as that would require the participants to trust that the source is impartial. It must be computable, so it cannot remain unpredictable forever, as it's obviously not unpredictable after it has been computed. If a protocol allows the participants to agree on an input to a random function without being able to compute the function until a later point in time, we can use that as a VRF. If all participants contribute something to the input, and each contribution alters the output in such a high degree that knowing all but one of the contributions, makes

one no better to guess the output than knowing one of the contributions, then a participant doesn't need to trust anybody but themselves to be convinced that the output is random.

With the exception of the explanation of random beacons, the rest of this section will introduce some known implementations of such a protocol.

2.2.2 Categories of VRFs

N-of-N secret commitments

A two-step protocol where N parties generate a random number that is only known after all parties reveal a secret. In the first round, participants generate a secret bitstring, calculate its hash, and publicly commit to the hash. In the second and last round, participants reveal their secret bits, i.e. the preimage of the hash they committed to. All the secret bits are aggregated, e.g. by XOR or concatenation, the result of the aggregation being the random seed. We see that the random seed will be unpredictable unless one knows the secrets of all the participants, and infeasible to guess if the secret bitstrings are of a minimum length. Assuming the secrets are chosen randomly, we will get a uniformly random bitstring as seed. This scheme is quite simple, but has some weaknesses. It can halt if one participant does not reveal their secret. Also, the last participant to reveal their secret can see the outcome before they reveal their own secret, and so choose to not reveal if the outcome is unfavorable. Participants can however be incentivized to cooperate, e.g. by losing reputation if failing to reveal their secret. This scheme can face scalability issues, as the risk of one participant going offline increases with the number of participants.

M-of-N secrets

A threshold multisignature scheme can be used to generate an unpredictable seed that will be revealed after a subset of participants sign a message. A multisignature scheme works so that the ability to sign a message is spread among N private keys. M is a threshold parameter which represents the number of private keys that are needed to sign the message in order to provide a valid signature. Each private key signing a message provides a signature share and a validity proof. A combining function takes M validity proofs and signature shares and outputs a valid signature and a proof, or it fails. By using the valid signature from the combining function as a random seed, we can make a verifiable random number that needs the cooperation of at least M members of a committee of N members.

We see that the random number will be unpredictable as long as there is no collusion of more than $M-1$ members. If we assume there is no such collusion, this scheme has several advantages from the above scheme. It will be able to provide a random number even if $N - M$ participants are unresponsive. When $M - 1$ participants have shared their signature share, any of the remaining signers can

privately see what the result is by generating the last share without sharing it. They could then avoid sharing their share if the random seed is unfavorable, but any one of them could continue the process by publishing their share. This is different from the above scheme in that any one of the participants won't be able to halt the scheme alone – unless $N = M$, of course. The main benefits with the M-of-N secrets scheme are therefore that at a collusion needs at least M participants to succeed, and the scheme will not be halted unless at least $N - M + 1$ users are unresponsive.

2.2.3 Delay functions

A delay function is an algorithm that is expected to take a predefined minimum amount of time to calculate. If the participants can agree on an input to the delay function whose output will be used as a random seed, it is guaranteed that nobody will be able to know the random seed before the minimum time of the delay function has passed. Some delay functions are probabilistic in that they only have an expected minimum time to compute. We let the variance of a delay function be the variance of the distribution of time elapsed when calculating the delay function multiple times, with uniformly random parameters if applicable. An ideal delay function is inherently sequential, has low variance, is moderately hard to compute, and is easy to verify.

To be inherently sequential, and therefore not parallelizable, is an important quality for delay functions, since a parallelizable delay function can be calculated arbitrary fast by an adversary with many processors. A delay function with high variance will, with unchanged computing power, sometimes be calculated very quickly and sometimes very slowly. This is a disadvantageous property, since a delay function that is too easy or too hard does not achieve its purpose. A delay function that is easy to verify saves verifiers from repeating the work of provers, and does not discourage verification as it is easy.

Delay functions are attractive for generating unpredictable numbers when a large amount of participants is involved. This is because all can contribute to an input, while only one party needs to compute the function in order to get the output. One disadvantage is that verifying the correctness of the output might require as much time as computing it. The delay function needs to be so hard that even an adversary with optimized hardware and software will not be able to compute it before a minimum time. If its use is to be trustless, each participant, even those with limited computing power, should be able to verify its correctness. While some constructions such as a simple hash chain do require as much time to verify as to compute it, there has recently appeared more complicated constructions of delay functions that require exponentially less time to verify than to calculate [32].

Example of using a delay function

Let's say we need a verifiable random number that should be unpredictable before some point in time t_{end}. In order to make sure no one is able to predict the random number before t_{end}, we say the random number is the result of a delay function, and we assume the delay function is a random oracle. By making assumptions of current hardware and algorithms, we estimate the delay function has a minimum calculation time of td_{min}. The delay function takes some input that we must agree on, e.g. a blockchain block hash that is not available before time t_{start}. As long as the input to the delay function is unknown by t_{start} and the assumption of minimum time needed to calculate the delay function holds, we can be certain that the random number is unpredictable by t_{end}. In this example we used a block hash, but any input that is unpredictable until t_{start} can be used, such as an aggregation of commitments from multiple participants.

Hash chain as delay function

A hash chain of a certain length used as a delay function is inherently sequential, has low variance, can be arbitrarily hard to compute, and is hard to verify. A hash chain is a sequence of hashes that are linked by each item being the hash of the previous item. A hash chain of length n and input h_0 is the composition of a hash function $Hash()$ n times which we call $Hash_n()$. What makes a hash chain sequential is that one cannot calculate $Hash_n()$ without knowing $Hash_{n-1}()$, and it cannot be calculated in parallel as one step's output is the next step's input. The time needed to calculate a hash chain is not probabilistic, so the variance of calculation time will be negligible. It can be made arbitrarily hard as the time needed to calculate a hash chain will increase linearly with its length. A drawback is that a verifier would need to calculate the entire hash chain in order to validate its correctness.

Proof-of-work as delay function

PoW is not sequential, has high variance, is arbitrarily hard to compute, and is easy to verify. Since PoW is parallelizable and has high variance, it's not really good as a delay function. An adversary can get lucky and calculate the result quickly, and thus avoid the delay. An adversary with multiple processors can calculate a PoW multiple times faster than an honest participant with one processor. Calculating a PoW can be seen as a probabilistic process where one makes attempts that result in success or failure. The probability of success for each trial is equal and independent of any other trial. If multiple parties tries to calculate the same PoW, but starting from different nonces, the distribution of time elapsed for each party will have high variance.

Weakly encrypted bits as delay function

Participants can weakly encrypt a secret by e.g. using a short encryption key, so that the secret is verifiable either after the time needed to brute force the decryption, or when the participant reveals their encryption key. This is discussed in [33], but is not completely satisfactory as a delay function, as breaking a secret encrypted with a short key is parallelizable. A scheme to combine a delay function with a weakly encrypted secret is suggested in [34]. The scheme works as follows: A secret s is encrypted with a strong key s. There is a function $W()$ that is easy to calculate if one knows a secret n but is inherently sequential to invert, and can be initialized with a parameter that says how many computations are needed to invert it. This can be used to generate unpredictable numbers by having participants commit to a commitment (c_1, c_2) where $c_1 := Encrypt(s, k)$ and $c_2 := W(k)$. We assume nobody has been able to find all k by brute forcing all $W^{-1}(c_2)$ within some time limit, when participants reveal their k and n. It must be verified that all $c_2 = W(k)$ before all s are calculated and aggregated to form a random seed. If a participant fails in revealing k, it can be calculated by eventually solving $W^{-1}(c_2)$.

Note that $Decrypt(c_1, k)$ will provide a value for all k, so we must verify that $c_2 = W(k)$. $W^{-1}(c_2)$ must therefore be guaranteed to have a solution that is possible to calculate within reasonable time. Otherwise, a single anti-social participant could halt the process by committing to a c_2 that takes one year instead of, say, one hour to brute force.

2.2.4 Random beacons

A random beacon is a service that broadcasts random numbers or provides random numbers on request, and can be used as a public source of randomness. The beacon can be considered a trusted third party in a transaction between untrusting parties, so that they all have a random number they can agree on. Rabin in [35] discuss the use of random beacons for transaction verification, but do not outline the design of a verifiable random beacon. Using sources of randomness that are hard to forge, such as financial data [36] and blockchain block hashes [37, 38] has been suggested, but is not considered safe from manipulation [39, 40]. The two organizations National Institute of Standards and Technology [41] and random.org [2] maintain public random beacons that are accessible over a web API, but their randomness is not verifiable and an application using it will have to trust these organizations.

2.2.5 Verifiable random oracles

Using random oracles for randomness often involves trusting the random oracle to act honestly. However, a random oracle can also use a random process that is

[2] RANDOM.ORG https://www.random.org/

verifiable. RanDAO [42] is a random oracle that can be invoked from an application that needs a random number. RanDAO uses an N-of-N commitment scheme, and will output the result of that process if all N players reveal their secret or an error otherwise. As discussed earlier, any one player in such a scheme is able to halt the process and force an error to be returned if they fail to reveal their secret. This makes it possible for an adversary to tactically not reveal secrets if the result is unfavorable. To prevent – or at least discourage – such behaviour, participants of RanDAO can be forced to make a deposit in order to join the process. The deposit will be confiscated if a player fails to follow the protocol. Such a deposit would have to be so high that the potential gains from halting the random process does not exceed the amount that is confiscated. If an actor uses an N-of-N commitment scheme with deposits that can be confiscated as a random oracle, they could to some degree trust that the random process will not be manipulated if there is at least one honest player – i.e. non-colluding player – that submits a uniformly random secret.

Such a scheme with deposits makes the assumptions that: (i) there is at least one honest, i.e. non-colluding player, and (ii) at least one honest player submits a uniformly random number. The former assumption is important because it makes sure that nobody can predict the outcome. All the players could be colluding and know each other's secrets and thus predict the outcome. The latter is also important, as it could be the case that players submit a secret that is not uniformly random, and that an adversary can use this fact to predict the output. To demonstrate this, consider if it were common that all players simply submitted a bitstring of zeroes as their secret. The process would succeed and will be verifiable, but the outcome would be predictable. The RanDAO scheme is spawned as a random oracle to be used by another application, and the players in the RanDAO don't necessarily care about producing a uniformly random number. Even if players are compensated for participating in the RanDAO, they would get the reward whether they submit a uniformly random number or a predictable number.

The issue of not being able to verify that secrets in RanDAO and similar schemes are actually random is addressed in [43]. A scheme similar to N-of-N secret commitments is used with the addition of a game with a reward mechanism that favors players submitting uniformly random numbers. For players maximizing their reward, the authors prove that a quasi-strong equilibrium in a game-theoretic sense exists when players submit uniformly random numbers. The scheme otherwise functions as a RanDAO that will either return a random number if all players follow the protocol, or an error otherwise. Players also have to deposit an amount so high that the potential gains of halting the process is lower than the deposit. A reward to players is paid so that any player playing the optimal game of submitting

uniformly random numbers gets an expected reward of at least $\frac{reward}{n\_players}$.

The above scheme allows one to spawn a random oracle that will produce a number that is uniformly random if players want to maximize their reward, and that is unpredictable as long as at least one player do not prematurely share their secret. The scheme still suffers from a principal-agent problem, in that even if the random process scheme, the agent, is executed correctly, it may not act in the best interest of the application that spawned it, the principal. An important assumption made is that players are non-colluding so that no single party knows all the secrets in the scheme. This is encouraged by rewarding players and having the process being open for participation by anyone who can make a deposit. The players will lose a large deposit by playing dishonestly, and the players will maximize their reward by using uniformly random numbers as secrets.

There is a problem when the potential gains from being able to predict the outcome is higher than what it costs to bribe players to reveal their secrets. The randomness of the scheme assumes players will act in a way that maximizes their reward, but players could potentially gain a higher reward by revealing their secret to the highest bidder, than what can be gained by playing honestly and compete for the in-game reward. This fact goes against the game theoretical aspects of the scheme, so that the scheme ultimately relies on at least one honest participant, as reward-maximizing participants will in some cases rather sell their secret if a market for secrets exists. And if the assumption is at least one honest participant, one might as well use a simpler and more flexible random oracle such as RanDAO with deposits and rewards. If selling the secret is not punishable, then there is not much to lose by selling it, as it could at most potentially make them lose their fraction of the reward, which can be lower than the payment for revealing their secret. The scheme suggested in [43] could be amended with a mechanism that confiscates a player's deposit if their secret is revealed by someone other than the player themselves, but this issue is not discussed in the aforementioned work.

2.3 Lotteries

Shamir et al. in [11] introduced a protocol for two parties playing virtual card games that involve randomness without a trusted intermediary. Playing so-called mental poker has been studied further in [14] where the authors suggest a protocol to play any mental game that will work between any number of players at least as there is an honest majority. Lotteries are also mental games that require randomness, but since they typically have many participants and are susceptible to sybil attacks, traditional mental poker protocols won't easily work.

Syverson in [33] and Goldschlag and Stubblebine in [15] suggested using data from sold tickets in a lottery as a source for entropy for a random function in

order to create a lottery with a verifiable random process. A lottery where only internal information is used in the random process is called *committed* or *closed*. Such internal information could be a number embedded in the ticket itself, chosen randomly by the ticket owner. Both their protocols require some time to elapse between when the last ticket is bought – which finalizes the input to the random function – and when the result of the random function is available. Goldschlag and Stubblebine employ a delay function that takes a minimum time to calculate to achieve this, while Syverson include a weakly encrypted secret in each ticket that is moderately hard, but not infeasible, to decrypt and use to calculate the random function.

The concepts of a closed lottery and using delay functions in the random process have been used by a number of similar schemes in subsequent works. Various enhancements such as privacy in [44] and more use of fair exchange in [45] have been suggested. A number of schemes use other means to generate randomness, such as verifiable random numbers generated by a committee of semi-trusted delegates in [16, 46, 47, 48]. Grumbach and Riemann in [49] use a random process inspired by distributed voting schemes – a Kademlia distributed hash table where each participant is a node who cooperates with other players in their subtree to generate verifiable randomness.

Although these schemes can generate randomness for a lottery without a trusted intermediary, they don't provide equally trustless solutions to the other aspects of organizing a lottery. One issue is simply to have the participants agree on who are actually participating. Most schemes solve this by having a lottery organizer digitally sign tickets so that the signature proves that a ticket is indeed valid. Should the lottery organizer refuse to pay the prize to a holder of a valid ticket, the ticket owner would have to give up or complain to some other authority. The problem of a dishonest organizer can be solved by storing the prize money with a bank or payment processor who issues tickets and prizes based on fair exchange. But by doing that, a bank or payment processor is involved as a trusted intermediary.

If the lottery is closed and information from the tickets are used as input to the random process, the organizer and participants need to agree on what is the correct set of tickets to be used. Since the organizer is the one who issues tickets, they could choose to not issue tickets during the last part of the purchase phase, so that the input to the random function is finalized before it is supposed to be. Doing so, the organizer has the power to choose what input to use for the random function, and possibly calculate the delay function before the purchasing phase is finished, so that the organizer can predict the result and purchase a winning ticket. This issue is identified in the literature, and can be handled in at least one of two ways. One possible way of handling it is to require that the organizer publish the

entire ledger of tickets continuously. It doesn't solve the issue completely, but it allows observers to see if suspicious behaviour is going on [33]. Another solution is to have a trusted intermediary who can also issue tickets if the organizer is unresponsive to requests [44]. These solutions do, however, necessitate some trust in either the organizer or another trusted intermediary, which is not desireable.

Syverson and Goldschlag et al. in [33, 50] discuss the topic of a *pari-mutuel* lottery concerning the organizer's ability to issue free tickets for themselves. If the lottery is pari-mutuel, meaning the entire prize fund comes from purchased tickets, the expected value of a ticket is independent on the number of tickets in existence, as the chance of winning decreases proportionally to the increased prize. But if the prize pool is larger than the cost of all tickets, the organizer could increase their expected value by forging tickets for themselves. The conclusion in [33, 50] is that a distributed lottery needs to be pari-mutuel unless it can manage sybil behaviour by the organizer.

A number of lotteries have appeared on various blockchains. Due to the openness and verifiability of blockchain transactions, anyone with some coding skills can create a lottery or verify a lottery's fairness on such a platform. Although creating a completely fair lottery on a blockchain platform is non-trivial, some claim to have accomplished the feat. SmartBillions [51] is a lottery on Ethereum that instantly pays out prizes. It's more akin to scratching tickets in that players play on their own without interacting with other players of the same game. One simply chooses a lucky number and finds out if one wins or not when the next block is mined. The randomness does, however, come from a block hash which is not considered secure for large prizes, as it can be manipulated by miners [39, 40]. FairLotto [52] is another lottery which is implemented on the Steemit blockchain. It works as a typical lottery in that participants contribute to a prize fund by buying tickets, and one winner is selected randomly to receive a share of the prize. It does, however, also rely on randomness from a source that can be manipulated by miners. It combines a secret the lottery organizer has committed to, and the transaction id of the last ticket bought to generate a random number that will be used to select the winner. In such a scheme the organizer can potentially bribe miners to tactically select a transaction to be the last ticket, so that a ticket owned by the organizer will win. There are other lotteries that use cryptocurrencies such as SatoshiDice [53], but their randomness is completely generated by the lottery organizer, so it's actually just an online casino that uses cryptocurrencies, and not a distributed lottery protocol.

There has also been interest in distributed lotteries on the blockchain in the academic literature. Andrychowicz et al. proposed a lottery implemented in Bitcoin transactions in [20]. The authors suggest that multiparty computation (MPC) on

a blockchain can enforce honest behaviour by having participants make deposits that will be confiscated if they fail to follow the protocol of the computation. This can potentially solve the problem of previous lottery schemes that eventually have to rely on a trusted third party to enforce payments and correct behaviour by the lottery organizer. Androchowicz et al. argue that traditional mental poker protocols cannot force participants to respect the outcome of the protocol, which might limit their use in practice. Since blockchains typically have a valuable cryptocurrency, funds can be deposited and later reclaimed on the condition that honest behaviour can be proven.

Following Andrychowicz et al., more Bitcoin based lotteries have been designed, including one concurrent by Bentov et al. [21]. These two schemes do, however, require a deposit that grows polynomially with the amount of participants. This makes a large lottery impractical, as prohibitively high deposits would be necessary to play the lottery securely. Bartoletti and Zunino in [23] and Miller and Bentov in [22] independently designed similar lotteries that require only a constant or zero deposit, respectively. These lotteries work by constructing a tournament of digital coin tosses where each participant is paired with an opponent in $log_2(N-1)$ *levels* for a lottery with N participants. Half the participants are eliminated in each level until there is one winner left who can claim the prize. This scheme does a trade-off by minimizing deposits, but increasing the number of levels from being $O(1)$ to being $O(log_2(N))$, which in turn increases the interactivity of the protocol.

2.4 Blockchain

A blockchain is an append-only data structure of cryptographically linked blocks. A blockchain has a first block of height 0 which is often called the *genesis block*. Each subsequent block of height h contains a cryptographic reference to the block of height $h-1$. New blocks are appended regularly, and the most recent block is called the *tip*. Blocks can contain a set of transactions, each of which represents a transition of a global state. The global state is implicitly defined by all the ordered transactions in the entire blockchain. Due to the links from a block to the previous block, one block cannot be altered without altering all subsequent blocks as well. A blockchain is typically widely distributed and blocks are hard to produce. This gives blockchains immutability and finality properties that make them fit for purposes such as transferring and storing value in a network of non-trusting participants.

A blockchain is commonly used as a global ledger that represents a state that defines ownership of various digital assets, these assets primarily being cryptocurrencies, but also property deeds and financial instruments [54]. Blockchain systems are also typically public and open for anyone regardless of their legal status or geographical location. A blockchain can be used for high value transfers and critical

computations because it is considered to have the properties of finality and liveness [55]. The term blockchain usually does not mean just the data structure, but also the implicit state represented in its transactions, the network of stakeholders interacting with it, and applications built on top of it. We will here consider the *blockchain network* to be the interconnected nodes running the full client software of the blockchain.

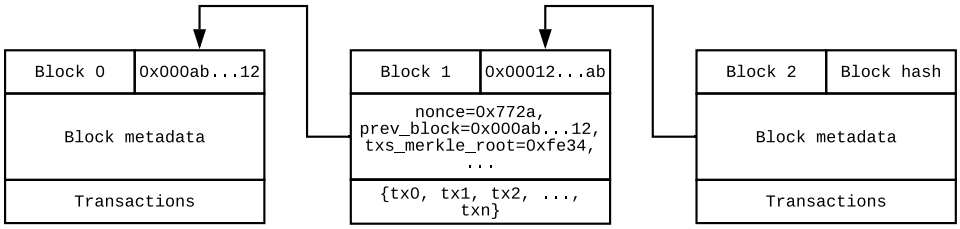


Figure 1: A chain of blocks linked with hashes.

2.4.1 Transactions

The purpose of a blockchain is to enable participants to reach a global consensus on an ordered list of transactions. Transactions are validated and embedded in blocks, and the set of all ordered transactions implicitly represents a state. In Bitcoin [56], the state that is represented is who has the authority to spend which coins. The state can be abstracted to represent a set of accounts with balances denominated in the currency *bitcoin*. A transaction can specify a short script that makes some coins spendable by executing a specific script with certain arguments – a spending policy for coins which usually involves a digital signature. Transactions can be abstracted to represent transfers of coins between accounts.

In the Bitcoin network, each transaction is applied to a state which results in an altered state. Transactions are validated before they are applied to the state. The process of validating transactions consists of multiple steps. First, the transactions must be of a valid format. Only a limited set of instructions can be used, and the total size of the script is limited to a maximum length. Second, the execution of a transaction's script must result in a specific output. Third, the transaction cannot spend more coins than it has authority to spend.

All coins in Bitcoin begin their life as a transaction output. Each block contains a *coinbase transaction* which defines some new coins, and the miner of that block decides how those coins can be spent. We assume the miner makes it so that the coinbase transaction coins can be spent by providing a signature with the miner's private key. The miner realizes this by defining the coinbase transaction's *output script* – a machine readable spending policy for those coins, which will be stored in

the block.

The output script is a list of operations that will be executed on the Bitcoin virtual machine (BVM). This virtual machine is stack based with a limited instruction set. While the coinbase transaction only has an output script, all other transactions, which we will call ordinary transactions, have an input script and an output script. The input script, as well as a reference to a previous transaction, must be defined in each ordinary transaction. When an ordinary transaction is validated, the output script of the previous transaction it references will be pushed to the stack of the BVM. Then the input script of the transaction will be pushed to the stack of the BVM, before the program is executed. If the stack at the end of the execution only contains a `true` value, the second step of the transaction validation process has succeeded.

In addition to the input script and a reference to a previous transaction, an ordinary transaction also contains a new output script and the amount of coins it spends. A valid transaction will move the coins it spends to a new output script. And so each transaction moves coins from one output script to another, and a coin's entire history will be stored in the blockchain.

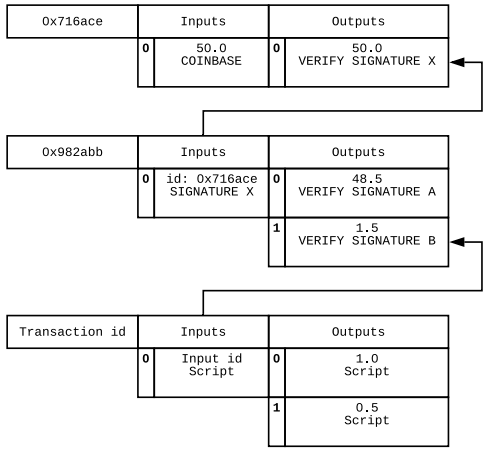


Figure 2: Transactions in Bitcoin.

2.4.2 Mining

Blocks are produced by miners who validate transactions and calculate a cryptographic puzzle that requires large amounts of computing power to be expended. The puzzles work as a probabilistic process where miners need to map a block's data to a small subspace of a function's range. Such a mapping constitutes a valid proof-of-work (PoW) and only blocks with a valid PoW will be accepted by other

nodes.

PoW is done to make producing blocks hard. Blocks being hard to produce has three features: First, the network will not be flooded with new blocks. Sybil attacks are common in unpermissioned networks, and it's hard to distinguish real users from sybil impersonators. Since PoW is easy to verify but hard to prove, performing a sybil attack is expensive. Second, it makes it possible to reach consensus. A blockchain can only have a single block of a specific height, so if there are multiple blocks of the same height, or even chains of blocks with the same height, the network can simply choose to accept the blocks with the most PoW as a straightforward way to reach consensus. Miners are rewarded for each block they produce, but only if it's accepted by the network. This ensures that miners will only be rewarded if they mine on the chain everybody consider to be the correct one. Third, it makes it difficult to rewrite blockchain history. Since each block is costly to produce, replacing an accepted block with another will be costly. This makes sure that the cost for an adversary to rewrite history is high, which again makes the network trust the blockchain's finality.

Since many miners are competing simultaneously, we will from time to time encounter a situation when two or more valid blocks are produced at about the same time. A miner who produces a valid block will broadcast it to other miners, who will in turn propagate it through the network. When miners receive a valid block of height h from a peer, they will start mining on top of that block to produce the block of height $h + 1$. If several blocks are mined and propagated through the network at about the same time, some miners will have different blocks of equal height, and will need to make a decision on which block to continue mining on top of. This situation is called a natural fork, as the chain is split at the end into more subchains. Eventually one of the subchains will be appended more than the other, and miners will accept whichever subchain is longest [3]. The shorter subchains will be abandoned and their transactions will not affect the state represented by the blockchain.

Small block reorganizations happen regularly as miners can mine separate blocks of the same height before either block has propagated through the entire network. This does not necessarily mean that any transactions are reversed, as the two blocks are likely to include quite similar transaction sets. Larger accidental block reorganizations can also happen during software releases that may contain bugs or conflicting consensus protocols [57], but this does not happen often on well-established blockchains.

[3] In Bitcoin, the subchain with the most PoW will be chosen, but unless the split is very large, this will always be the longest subchain.

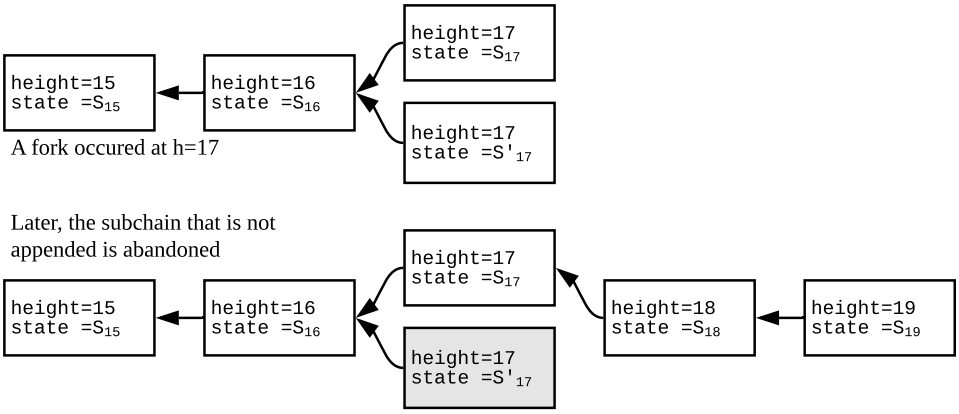


Figure 3: A natural blockchain fork.

2.4.3 Blockchain threats

Since Bitcoin's inception there has been discussions on the security and threat models to blockchains. Much of the discussion involves the role of miners and their ability or inability to control the network. While there has been few unsuccessful attacks on the larger blockchains such as Bitcoin and Ethereum, the threat models and assumptions need to be continuously reconsidered for new use cases of the blockchain.

Block reorganization

During the situation of several subchains being on the network, one view of the blockchain might give a different state than another view. Say one observer had a view with subchain s_1 that included transaction t_1, and another observer had a view with subchain s_2 that did not include transaction t_1. If s_1 is eventually abandoned, as miners continued to mine on top of s_2, then t_1, which seemed to be included, is no longer part of the blockchain. This phenomenon is called a block reorganization, and can potentially reverse transactions.

A block reorganization violates the immutability property of the blockchain as blocks can be removed. Since miners have an interest in mining on the same chain, as only one chain's coins will be valuable, the network tends to reach consensus on which chain is correct quite quickly. A common heuristic for the Bitcoin blockchain is to consider transactions in blocks that have at least 6 blocks appended to them as final, which takes on average one hour to happen. As a general rule, the more blocks are mined on top of a block, the stronger its immutability.

A block reorganization attack can be conducted by an adversarial miner or collusion of miners who aims to replace a larger section of the most recent blocks in

order to alter the state. If blocks are replaced, transactions can also be reversed. A block reorganization attack can be used to perform a *double spend*, where a purchaser makes a payment that is reversed after they receive the goods for the payment. Such an attack is done by secretly mining an alternative subchain while waiting for a certain state to be reached on the main blockchain, and then broadcasting the secret subchain in hope that the network will accept the adversary's alternative subchain that is more favorable to the adversary, i.e. the payment transaction is reversed. This allows the adversary to trick the network into believing the blockchain is in a certain state, while it will actually change once they broadcast the secret subchain. For the secret subchain to be accepted by the network, it needs to be longer than the public chain. Since it's difficult to produce blocks, this can only be done successfully on average if the adversary controls more than 50% of the mining power or power to produce blocks. As a consequence, block reorganization attacks is often interchangeably used with the term *51% attack*.

Censorship

Users can change the state of a blockchain by broadcasting transactions that will be included in a block by a miner. Due to limited space in blocks on blockchains like Bitcoin and limited computational resources on blockchains like Ethereum, there is a limited amount of transactions that will be included on the blockchain. Transactions can optionally include a fee that is paid to the miner who includes the transaction in their block, and the size of this fee is the basis on which transactions miners choose to include. If miners are merely maximizing transaction fees, such a pricing mechanism for blockchain resources maintains the liveness and openness properties of the blockchain, as there is a single objective criterion for participating.

Miners do, however, have the power to discriminate transactions on any other basis than fees as well, e.g. on a political or self-interest basis. If a miner refuses to include transactions from a specific user, then that user will only have their transaction included if another miner includes it. If a collusion of miners all agree to not accept transactions from a specific user, and the collusion controls a sufficiently large portion of the mining power, they can be able to effectively censor that user. Even if a non-colluding miner eventually includes the censored transaction in a block, the collusion can choose to ignore that block. Such a situation will violate the openness property, as the blockchain is no longer accessible by anyone.

Unlike a block reorganization attack, which affects the finality of many transactions, a censorship attack can be targeted at a single user. One of the security assumptions commonly made in blockchain systems is that it is not in miners' interest to launch attacks that threaten the main security properties of the system, as miners are typically heavily invested in the cryptocurrency through e.g. capital investments [58, 59]. An attack that affects many users might trigger a backlash

by other stakeholders and thus hurt the miners who launched the attack. Since a censorship attack can target single users, it might have a lower disruption cost, i.e. cost to launch the attack, than the disruption cost of a generic 51% attack. So even if a blockchain is secure from some types of attacks, it might not be secure from other attacks such as a censorship attack. When building applications on top of a blockchain, it is very important to consider which types of attacks the application is vulnerable to.

Selfish mining

Selfish mining is a type of miner behaviour that might enable some censorship by a collusion with as little as 25% of the mining power [60]. While an altruistic miner will mine on top of whichever block they know about with the greatest height, a selfish miner will bias their mining towards their own blocks. A selfish miner trying to mine a block of height h will not immediately start to mine on the block of height $h+1$ when receiving a valid block of height h. Instead, they will continue trying to mine their own block of height h until they receive a block of height $h+k$ for some k, by when they start the same procedure again.

If the selfish miner controls a small ratio of the mining power, such a strategy will result in loss of potential profits, as they will waste resources on subchains that will be abandoned. As the ratio of their mining power goes up, their reluctance to accept blocks from other miners might cause the network to also abandon blocks from other miners at a more frequent rate than what their respective mining power would say. This will in turn incentivize other miners to more readily accept the selfish miner's blocks, as this will decrease the chances of their blocks being abandoned and the block reward being lost.

If such a scenario plays out like that, the selfish miner can also choose to censor a certain type of transaction and force the network to join in on the censorship policy. The selfish miner enforces this by refusing to accept any block that contains a transaction of the type they censor. If the subchain the selfish miner favors is more likely to eventually be accepted, then other miners will produce blocks they are sure will be accepted by the selfish miner.

2.5 Trustworthy computing

2.5.1 Smart contracts

Smart contracts is the concept of using computer code to enforce and secure contract law and specify other formal relationships and expectations in society [61]. Digital signatures make it possible to authenticate an owner of digital assets. A system with an authentication mechanism and a record of digital assets that can be transferred between owners can serve as a smart contract platform. A smart

contract is similar to any computer program in that it will execute some code, but it is different in that the execution and the result of it represent some legal concept, such as the right to vote, the ownership of something, an identity, or a more complex construct. A smart contract is not necessarily legal in that its code has legal status is any jurisdiction. It is rather that the smart contracts exist in an environment where they can enforce something, or the meaning of their computation represents something enforceable. In such a computer environment, legal entities can be authenticated with digital signatures, and agreements, expectations, and conditions can be written in code.

A simple example is a trust fund constructed in Bitcoin smart contracts. A grantor wants to make a trust fund for a beneficiary. The grantor programs a series of Bitcoin transactions $\{tx_0, tx_i, ..., tx_n\}$ so that each is spendable on the conditions that a) the transaction is digitally signed by the beneficiary, and b) i years have passed since the trust fund was created. If the smart contract platform support sufficiently expressive scripts, it is easy to imagine more complicated constructs of a smart contract. The trust fund contract could be extended with almost arbitrary conditions, such as allowing spending if a digital certificate of hospitalization or an acceptance letter from an educational institution is provided. In this way a legal construct can be created and enforced without the need of a code of law or a state apparatus to enforce the law.

2.5.2 Ethereum

Ethereum [62] is a platform for for deploying and executing smart contracts. Ethereum has a virtual machine (EVM) with a Turing-complete instruction set and a key-value store which forms a global singleton state. Smart contracts can be executed by broadcasting signed messages called transactions to a network of nodes. Transactions and the code they execute are processed in order by a global network of mining nodes who run the EVM and process transactions they receive. Transactions are stored in blocks which form a blockchain and are considered immutable after some point in time. In addition to executing existing smart contracts, transactions can also deploy new smart contracts or transfer a cryptocurrency called *ether* from one account to another. Even though the EVM is Turing-complete, each smart contract can only perform a maximum amount of instructions decided collectively by miners and measured in a unit of account called *gas*. Altogether, this forms a replicated byzantine fault-tolerant key-value store with an integrated currency with a general scripting language, open to anyone.

Transactions

Transactions are the basic units of communication in the Ethereum system. Miners validate each transaction before they include it to a block where the transaction's

actions cause a state transition in the global state. Such a state transition may represent cryptographic secure ownership of titles, funds, tokens and more, depending on the transaction itself and the participants involved in it. Transactions always contain an address which identifies an *account*, and may contain an optional payload of data which is the arguments to a function in a smart contract, and can also optionally transfer the cryptocurrency *ether*.

A special kind of transaction is the one that transfers ether from one account to another. It's different from other transactions in that it does not need the EVM to be processed, as it simply updates ether balances. Another special transaction is the contract creation transaction. This transaction includes a payload of bytecode which defines a new smart contract, and if successfully executed, such a transaction will permanently store the contract with a newly created associated account on the blockchain.

Accounts

Accounts are agents in the Ethereum network. All transactions must be initiated by an account, and each account has a non-negative balance of the cryptocurrency ether. There are two types of accounts: contract accounts (CA) and externally owned accounts (EOA). A contract account is a deployed smart contract with its associated bytecode. A CA can receive and send ether and invoke other smart contracts, and its behaviour will depend entirely on its code. An EOA is addressed and represented by the public key of an asymmetric key pair. An EOA can receive and send ether and create transactions, and all transactions it makes must be signed with the private key of the account.

Ether

Ether is a cryptocurrency integrated into Ethereum. New ether is created in each new block and similarly to the block rewards in Bitcoin, it is distributed to miners in order to incentivize honest behaviour. All ether in existence is associated with an account, and can be spent in one of two ways. For an EOA, only a valid signature from the account's private key is needed to spend the ether associated with the account. CA can also send ether, but the spending conditions can be arbitrarily programmed in the smart contract's code, which allows for a huge variety of spending policies. Ether is also the only currency that can be used to pay for transaction fees, which are measured in another currency internal to Ethereum called gas.

Gas

The cost of having transactions recorded on the blockchain is denominated in gas. Each computation performed in the EVM, each byte added to the global key-value store, and each transaction validation has a fixed gas price. The prices are carefully designed to reflect the effort miners consume by processing the transaction.

E.g. creating a new contract results in bytecode being permanently stored on the blockchain and so has a high gas usage, while a simple bitwise operation on two word-sized operands does not require much effort and so has a low gas usage.

Due to the impossibility for miners to accurately estimate the effort needed to process a transaction by only inspecting it, each transaction includes an amount of gas $start\_gas$ it's willing to spend. When a miner processes the transaction, each instruction performed will subtract some gas from this amount. If $start\_gas$ is not enough to pay for an instruction during processing, the transaction is reverted and its state transition is not recorded. The maximum amount of gas that can be included in a single transaction is a global variable collectively set by miners.

Mining and gas price

Gas is a unit of account for resources on the EVM and a virtual currency within Ethereum. It can only be bought, and can only be bought with ether. In addition to single transactions having a maximum gas limit, entire blocks also have a dynamic maximum gas limit collectively set by miners. The sum of gas usage of all transactions in a block cannot exceed this limit, and it sets a limit on the number of transactions that can be processed in a single block.

In addition to the maximum amount of gas $start\_gas$ its sender is willing to spend, a transaction includes a price per gas unit gas_price denominated in ether. Enough ether to cover the maximum gas usage $start\_gas \cdot gas\_price$ is deducted from the sender's account when the transaction is being processed. If there is remaining gas when the transaction is completed, ether corresponding to the remaining gas is refunded to the sender. Ether corresponding to the gas spent is transferred to the miner of the block that contains the transaction.

The gas price is used to incentivize miners to process transactions. A higher gas price will cause miners to process a transaction quicker, while a transaction with a low gas price might never be processed. This mechanism forms a market between senders of transactions and miners where senders bid a gas price which miners might accept.

Contract development and deployment

Smart contracts are added to Ethereum by deploying a contract's bytecode in a special transaction. A deployed contract will always have an address and also a namespace in the global key-value store called the contract state, where values can be persistently stored. All executable code in contracts exist in functions that are referenced in a hash map stored in the contract's namespace.

Contracts are typically programmed in a high level language and compiled to bytecode which is readable by the EVM. A developer environment usually includes a local blockchain or a global alternative Ethereum blockchain called a testnet

where ether is not scarce, so that contracts can be tested without needing to spend real valuable ether.

Decentralized applications

Ethereum often serves the role of a secure backend for applications accessed through a frontend web interface. The frontend typically has the ability to read the Ethereum state and make transactions to specific contracts. Such a system of one or more smart contracts and a user interface is commonly called a *dApp*, short for decentralized application. A dApp is in many ways similar to other applications on the web, but a dApp using Ethereum has an immutable backend with value transfer and smart contract capability included.

3 Implementation

3.1 Tournament lottery outline

3.1.1 Digital coin toss

A coin toss is a random process in which a binary outcome is decided. While a physical coin toss is decided by a coin landing on either the heads or tails side, a digital coin toss is determined by a function with range $\{0, 1\}$. In a coin toss in the usual sense, the two outcomes are equally likely. While either outcome can be biased to an almost arbitrary bias, all digital coin tosses in this thesis will have equally likely outcomes.

We consider the case where two untrusting parties want to arrive at an unpredictable outcome that is verifiable. One way of achieving this is using a two round protocol as described in [12] where each party commit to a hash in the first round, and then reveal the preimage in the second round. If we assume that a secure hash function is used, then either party can see the commitment of the other player without being able to guess what the preimage is. When the preimage, or secret, is revealed in the second round, either party can verify that the secret is actually the preimage of the commitment, thus making the protocol verifiable. The coin toss is performed by doing an operation, such as XOR, with both the parties' secrets as operands. The result of this operation is completely unpredictable to either party, as they do not know their opponent's secret. For a 50-50 coin toss, we can simply decide the outcome by something simple as whether the least significant bit of the result is 1 or 0.

A secure implementation of this protocol would require the parties to mix in some salt when hashing the secret, i.e. by concatenating the secret with a predefined string. This is to prevent one party from committing to the same value as their opponent, so that the preimages are equal to each other. That would result in the outcome being predictable, i.e. always 0 if we use the XOR operation.

Say we have Alice and Bob conducting a digital coin toss with commitments c_A and c_B and secrets s_A and s_B. The salt is a predetermined string both parties use $salt$, and the hash function is $Hash()$. The commitments are calculated as $c := Hash(salt, s)$, and are verified by $c = Hash(salt, s)$. We get the result r of the XOR operation $r = s_A \oplus s_B$. The outcome is then determined by if $r \ mod \ 2 = 0$, then Alice wins, or if $r \ mod \ 2 = 1$, then Bob wins.

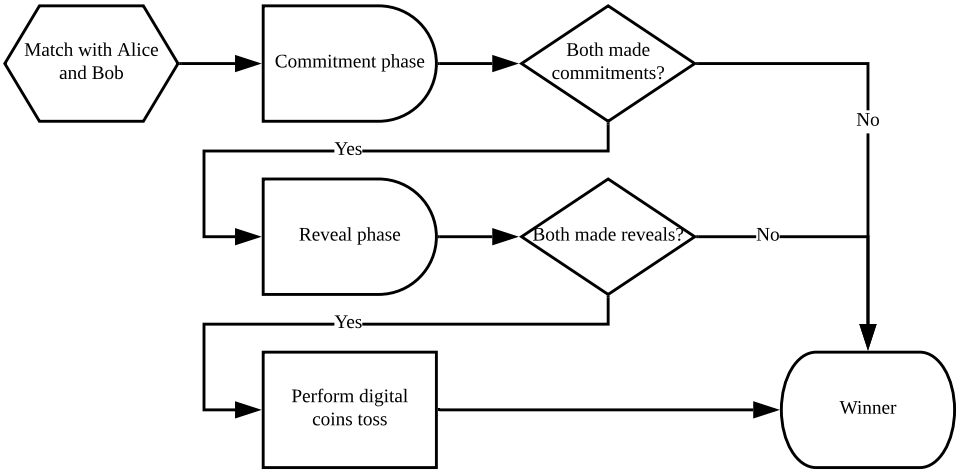


Figure 4: Flowchart of a digital coin toss.

Tournament of coin tosses

A digital coin toss can determine a winner out of two players, but can we use the same mechanism to determine a winner when there are more than two participants? As outlined in [22, 23], we can construct a tournament of matches where the winner of each match is determined by a digital coin toss. The tournament can be represented as a full binary tree where the root node is the final match, whose winner is the winner of the entire tournament. Winning any other match will make one advance to a match one step closer to the final match. The players of each match in an internal node, which we call *internal matches*, are determined by the winners from each of its two children. Participants in the tournament are represented as an ordered set, and each participant is assigned as a player to exactly one leaf node match.

Such a tournament will have $N = 2^L$ players where L is the height of the binary tree representing the tournament. There will be $\frac{N}{2}$ leaf node matches and $N - 1$ matches in total. We refer to matches with the same height in the binary tree as belonging to the same *level* in the tournament. Matches with the same level can be played concurrently and independently of each other, but matches in internal nodes of level l cannot be played before all matches in level $l - 1$ are determined.

It may be the case that one or both players in a match fail to complete the digital coin toss in a match. Matches of each level in the tournament will be initialized with global time limits for when players have to have completed a certain procedure, such as making a commitment or a reveal. If only one of the players fails to e.g. reveal their secret before the relevant time limit, that will be interpreted as

a forfeiture, and the other player will win the match. If both players fail to complete the same procedure, e.g. if neither player makes a commitment before the commitment time limit, a default winner will be selected to ensure each match has a defined winner by the last time limit. The default winner behaviour makes sure that the lottery will always end up with a defined winner, but will not affect an honest player who follows the coin toss protocol.

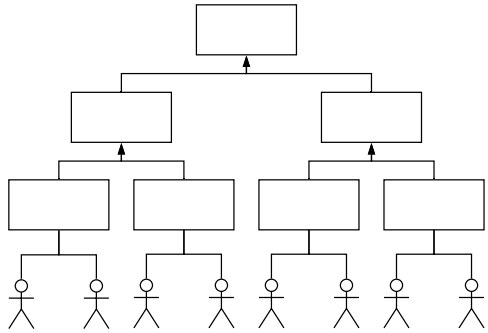


Figure 5: Tournament tree of 8 players.

3.1.2 Note on the lottery being non-deterministic

Why the tournament is not deterministic

It's worth taking a moment to consider the implications of the timeout conditions in the tournament. Ideally, both players in each match will perform the digital coin toss. Since the secrets in the coin toss are committed to, if we assume both secrets will be revealed, there is just one possible outcome of the match. However, if either one player fails to reveal their secret by the timeout, then the other player will win. The consequence of this being that a match is not deterministic when commitments are made. Only when both players have revealed their secrets or the timeout happens can the result be determined.

This means that even if all players commit to secrets for each level of the tournament prior to playing a single match, any player can actually end up winning independently of the secrets, as the outcome can depend on who times out in which matches. Note that if we assume that all players will perform the coin toss successfully, then the tournament is completely deterministic once all commitments have been made, and there is just one single valid winner for that particular ordered set of players and commitments. If it were the case that no timeouts would happen, we could even use a single commitment and secret from each player that would be used for each match at every level of the tournament. If players can choose whether to reveal their secret or not, we cannot reuse secrets from one match to another, as will be demonstrated below.

How to exploit a non-deterministic tournament

In a practical implementation of the tournament, we would have to have timeouts to avoid the protocol halting. We must also assume that players can be colluding. In a collusion, each player would know each other's secret, so they would know who would win against whom. If we simply used the same secret for matches in all levels, then even if the result is unpredictable before all players have revealed, a collusion could wait for all the non-colluding players to reveal, then tactically choose which players of the collusion should reveal, so that they are guaranteed to win matches in subsequent levels. Because of this, unless we can assume no collusion, we must require players to commit to different secrets in each match, and by that make the protocol more interactive.

3.1.3 Phases

We implement a lottery that uses a tournament of coin tosses to fairly determine the winner. It will be implemented on Ethereum where smart contracts are an important logical component. Our lottery is a system made out of smart contracts that interact is specific ways that will be explained below. The lottery has a lifecycle with separated phases, so it will be described according to those.

Set up phase

The tournament will be implemented as separate match contracts that each represent a match in the tournament tree. The matches are initialized with data that allows them to determine who its players are. Internal matches are initialized with two references to contracts in the previous level of the tournament, and it will call a `getWinner()` method from these contracts to determine who its players are. Leaf node matches, which we will call *first level matches*, are initialized with an index that corresponds to a player in the ordered set of players. The leaf node matches have a reference to a *master contract*, which they will call to retrieve the players that are assigned to the match.

While the match contracts form a tournament on their own, we need a contract to keep track of the state of the lottery and handle things such as taking deposits, maintaining the set of participants, and verifying who can claim the prize. This is the responsibility of the master contract. The master contract is initialized with the number of participants the lottery can have, which must be a power of two, a price participants must pay in order to join, and a start time t_{start}. To be able to verify the winner of the tournament, the master contract needs a reference to the final match contract. The entire lottery is set up by first deploying the master contract without a reference to the final match, as it has not been deployed yet. The match contracts will be deployed level for level, starting with the first level contracts. When the final match is deployed, the lottery will be initialized by setting the final match in

the master contract.

Deposit phase

Once the master contract and all the match contracts are set up, any potential player can verify that the contracts are set up correctly by inspecting each contract. Players can then join the lottery by sending a `deposit()` transaction that sends ether equal to the ticket price to the master contract. Making a deposit will increase the master contract's balance and insert the sender into a list of players, so that players are ordered by when they joined. In a lottery of N players, the list of players will be indexed i_{player} from 0 to $N-1$. Each first level match is initialized with a unique index i_{FLM} from 0 to $\frac{N}{2}-1$. A first level match retrieves its players by looking up $i_{FLM} \cdot 2$ and $i_{FLM} \cdot 2 + 1$ in the match contract's list of players.

Deposits will be possible as long as the lottery is not full and a start time is not yet reached. The lottery is full once the amount of deposits has reached the predefined number of participants. If the lottery is not full by the start time, it will not be able to start, and players can withdraw their deposits. If the lottery is full by the start time, it moves into the next phase.

Playing phase

Each leaf node match contract should have a time t_{commit} equal to the start time of the lottery, by when players can make commitments to the digital coin toss. The time limit t_{reveal} is some time later when players can make a reveal transaction with their secret which is validated in the match smart contract. The last time limit of a match contract is t_{play} by when it's no longer possible to reveal and the match is determined. When the match is determined, the method `getWinner()` will return a player address. If the current time is less than t_{play}, this method will raise an error. If there is a stalemate in the match, i.e. neither player makes a commitment and the t_{play} limit is passed, the `getWinner()` method will return a default winner. The default winner is the first player in the match, i.e. the left player in the tournament tree.

Internal match nodes should have a t_{commit} equal to the t_{play} limit of matches in the level below. As mentioned, each internal match has a reference to two matches from the previous level, in which the two winning players are eligible to play the internal match. This way the tournament proceeds level by level with time limits defined at contract initialization. Players will need to make transactions to the match contracts which validate the sender and the inputs. The final match will be played just like any other match, but the master contract will have a reference to this match, so that it can validate the winner of the tournament.

Withdrawal phase

Players can send a transaction to the `withdraw()` function of the master contract. If the final match has a winner, it will transfer the balance of the master contract to the sender, after validating that the sender is the same account as the winner. If the t_{start} time limit has passed and the lottery is not full, the lottery did not get enough players to start. In this case, the `withdraw()` method will return the deposit to the sender if they have made a deposit.

3.2 Code

The only smart contracts used in the lottery are the master contract and the two types of match contracts – first level match contracts and internal match contracts, all of which are implemented as Solidity programs. Solidity [1] is a high level language that compiles to EVM bytecode. Deploying a smart contract on Ethereum is done by making a special deployment transaction that contains the contract's compiled bytecode. A deployed contract is identified by a unique address that is assigned when the deployment transaction is made.

Setting up the lottery is a multi-step process that must happen in a specific order. In order to manage this process, we have made a script to automate it. The smart contracts and deployment scripts also have associated unit tests and a simulation suite.

The listings included in this chapter have been truncated somewhat by removing constructors and comments, and are not necessarily compilable code. See the Github repository [2] and Appendix A for full working code.

3.2.1 Master contract

`LotteryMaster` is the smart contract responsible for registering participants, taking deposits, and distributing the prize. It serves as a central hub in the lottery that all players must interact with. Whoever deploys this contract is the lottery organizer, who will be the only account with the authority to initialize the lottery by setting the final match.

The constructor of this contract takes the number of participants `N`, the price of participation `price`, and a start time `tStart`. It is only possible to make a deposit after the final match of the lottery is set with `setFinalMatch()`. The `finalMatch` field is a reference to a match contract which is the apex of the tournament tree of matches. This field cannot be set on deployment, as the matches in the tournament need a reference to a master contract, and the master contract needs a reference to a match contract. Due to this circular dependency, either all the matches of the

[1] Solidity v0.5.0 https://github.com/ethereum/solidity
[2] https://github.com/viktorfa/lottery-truffle

first level or the single master contract need a two-step initialization. Since the gas usage of a single transaction is less than that of many, we choose to have the two-step initialization in the single master contract.

```
contract LotteryMaster {

    address[] public players;
    mapping(address => uint256) public deposits;
    AbstractLotteryMatch public finalMatch;
    uint256 public nPlayers;

    address public owner;
    uint256 public price;
    uint256 public N;
    uint256 public tStart;

    bool public isInitialized;
    bool public isFull;

    function setFinalMatch(AbstractLotteryMatch _finalMatch) public {
        require(msg.sender == owner);
        require(finalMatch == AbstractLotteryMatch(0));
        finalMatch = _finalMatch;

        isInitialized = true;
    }

    function deposit() public payable {
        require(block.number < tStart);
        require(isInitialized == true);
        require(msg.value == price);
        require(isFull == false);
        require(deposits[msg.sender] == 0);

        players.push(msg.sender);
        deposits[msg.sender] = msg.value;
        nPlayers++;

        if (nPlayers == N) {
            isFull = true;
        }
    }

    function withdraw() public {
        if (block.number >= tStart && !isFull) {
            msg.sender.transfer(deposits[msg.sender]);
        } else {
            address lotteryWinner = finalMatch.getWinner();
            require(msg.sender == lotteryWinner);
            msg.sender.transfer(address(this).balance);
        }
    }

    function getPlayer(uint256 index) public view returns (address
        player) {
        player = players[index];
    }
```

```
53 }
```

Listing 3.1: Lottery master contract

3.2.2 Match contract

A match contract implements a digital coin toss between two players. The digital coin toss determines a winner either by the value of the least significant bit of the result of an XOR operation on the two players' secrets, or by one or both of the players failing to perform the procedures in the contract by some predetermined time limit. Due to slightly different behaviour between matches of the first level of the tournament and matches in internal nodes of the tournament, we split the code between two smart contracts who both implement an interface from an abstract smart contract.

A match has two players referenced by address in the `alice` and `bob` fields. By a start time `tCommit`, the players can set a commitment of bits with the `commit()` method. Another time limit is set in `tReveal` which is when making commitments is no longer possible, but revealing the secret committed to is possible. This is done with the `reveal()` method which verifies that the secret is the preimage of the commitment, and stores the secret in persistent storage. The final time limit is `tPlay`, after which players can no longer reveal secrets. After the final limit, the outcome of the match is guaranteed to be determined.

The `getWinner()` method performs the digital coin toss if both players have revealed their secrets. If either player has performed more steps than the other player, e.g. made a commitment while the other has not, the player who has done the most steps will win. If neither player has revealed their secrets, and no player has performed more steps than the other, then the winner of the match will be `alice` by default.

A match has a way to determine which players are eligible to play in it. This is handled differently in `FirstLevelMatch` and `InternalMatch`. The former is initialized with an index and a reference to a `LotteryMaster` contract. Each player in the master contract is indexed so that each player is paired with another who is their opponent in a match of the first level. The latter is initialized with a reference to two matches, `left` and `right`, of the previous level in the tournament. All matches will return their winner in the `getWinner()` method, which the internal match uses to determine its players.

```
1  contract FirstLevelMatch is AbstractLotteryMatch {
2
3      address public alice;
4      address public bob;
5
6      mapping(address => bytes32) public commitments;
7      mapping(address => uint256) public secrets;
```

```
8
9       LotteryMaster public lottery;
10      uint256 public index;
11
12      uint256 public tCommit;
13      uint256 public tReveal;
14      uint256 public tPlay;
15
16      function commit(bytes32 _c) public {
17          require(tCommit < block.number);
18          require(tReveal > block.number);
19
20          alice = lottery.getPlayer(index * 2);
21          bob = lottery.getPlayer(index * 2 + 1);
22          require(msg.sender == alice || msg.sender == bob);
23          require(commitments[msg.sender] == 0);
24
25          commitments[msg.sender] = _c;
26      }
27
28      function reveal(uint256 _s) public {
29          require(tReveal < block.number);
30          require(tPlay > block.number);
31
32          require(keccak256(abi.encodePacked(msg.sender, _s)) ==
                commitments[msg.sender]);
33
34          secrets[msg.sender] = _s;
35
36      }
37
38      function getWinner() public view returns (address winner) {
39          require(tPlay < block.number);
40
41          if (alice != address(0) && bob == address(0)) {
42              return alice;
43          } else if (alice == address(0) && bob != address(0)) {
44              return bob;
45          } else if (alice == address(0) && bob == address(0)) {
46              return lottery.getPlayer(index * 2);
47          }
48
49          if (commitments[alice] != 0 && commitments[bob] == 0) {
50              return alice;
51          } else if (commitments[alice] == 0 && commitments[bob] != 0) {
52              return bob;
53          } else if (commitments[alice] == 0 && commitments[bob] == 0) {
54              return lottery.getPlayer(index * 2);
55          }
56
57          if (secrets[alice] != 0 && secrets[bob] == 0) {
58              return alice;
59          } else if (secrets[alice] == 0 && secrets[bob] != 0) {
60              return bob;
61          } else if (secrets[alice] == 0 && secrets[bob] == 0) {
62              return lottery.getPlayer(index * 2);
63          }
```

```
64
65          if ((secrets[alice] ^ secrets[bob]) % 2 == 0) {
66              return alice;
67          } else {
68              return bob;
69          }
70      }
71  }
```

Listing 3.2: Lottery first level match contract

```
1   contract InternalMatch is AbstractLotteryMatch {
2
3       address public alice;
4       address public bob;
5
6       mapping(address => bytes32) public commitments;
7       mapping(address => uint256) public secrets;
8
9       AbstractLotteryMatch public left;
10      AbstractLotteryMatch public right;
11
12      uint256 public tCommit;
13      uint256 public tReveal;
14      uint256 public tPlay;
15
16      function commit(bytes32 _c) public {
17          require(tCommit < block.number);
18          require(tReveal > block.number);
19
20          alice = left.getWinner();
21          bob = right.getWinner();
22          require(msg.sender == alice || msg.sender == bob);
23          require(commitments[msg.sender] == 0);
24
25          commitments[msg.sender] = _c;
26      }
27
28      function reveal(uint256 _s) public {
29          require(tReveal < block.number);
30          require(tPlay > block.number);
31
32          require(keccak256(abi.encodePacked(msg.sender, _s)) ==
                commitments[msg.sender]);
33
34          secrets[msg.sender] = _s;
35      }
36
37      function getWinner() public view returns (address winner) {
38          require(tPlay < block.number);
39
40          if (alice != address(0) && bob == address(0)) {
41              return alice;
42          } else if (alice == address(0) && bob != address(0)) {
43              return bob;
44          } else if (alice == address(0) && bob == address(0)) {
45              return left.getWinner();
46          }
```

```
47
48          if (commitments[alice] != 0 && commitments[bob] == 0) {
49              return alice;
50          } else if (commitments[alice] == 0 && commitments[bob] != 0) {
51              return bob;
52          } else if (commitments[alice] == 0 && commitments[bob] == 0) {
53              return left.getWinner();
54          }
55
56          if (secrets[alice] != 0 && secrets[bob] == 0) {
57              return alice;
58          } else if (secrets[alice] == 0 && secrets[bob] != 0) {
59              return bob;
60          } else if (secrets[alice] == 0 && secrets[bob] == 0) {
61              return left.getWinner();
62          }
63
64          if ((secrets[alice] ^ secrets[bob]) % 2 == 0) {
65              return alice;
66          } else {
67              return bob;
68          }
69      }
70  }
```

Listing 3.3: Lottery internal match contract

3.2.3 Lottery setup code

The lottery setup code is written in JavaScript [3], an interpreted language that runs in all modern web browsers and in the NodeJS runtime environment [4]. Since it's common to make web clients for smart contract applications and modern web apps are usually programmed in JavaScript, that language is commonly used in the layer between user and the blockchain API. The Truffle framework [5] contains a library that provides a useful abstraction to smart contracts and the Ethereum RPC client, and includes tools for handling the development lifecycle of smart contracts. web3.js [6] is a comprehensive library to interact with Ethereum and the EVM in JavaScript. Ganache [7] is an Ethereum blockchain with an RPC client that can be installed locally when developing smart contracts. Eth Gas Reporter [8] is a tool that measures the gas used for each transaction during a Truffle test suite, and was used to collect data on gas usage.

In order to set up a valid lottery, the master contract and match contracts need to be initialized with correct parameters. In addition to testing and simulation, this is the main purpose of the lottery setup code. The lottery is set up by first deploying

[3] JavaScript (ECMAScript 2018) https://github.com/tc39/ecma262
[4] Node.js v10.7.0 https://github.com/nodejs/node
[5] Truffle v5.0.15 https://github.com/trufflesuite/truffle
[6] web3.js v1.0.0-beta.37 https://github.com/ethereum/web3.js
[7] Ganache CLI v6.4.3 https://github.com/trufflesuite/ganache-cli
[8] eth-gas-reporter v0.2.0 https://github.com/cgewecke/eth-gas-reporter

a master contract, then deploying, level for level starting from the first level, match contracts that constitute a valid tournament tree, and finally initializing the master contract with the final match of the tournament tree. The setup code finds the correct parameters for time limits, indices of first level matches, and left and right addresses for internal matches.

The simulation code consists of two test suites. The first does the routine a lottery organizer needs to do in order to set up a lottery, which is creating and deploying the master contract and all match contracts. The second does all of what the first does in addition to simulating the lottery being played. In the second suite, players join by making deposits and then play each level of the tournament until one player is left as winner and withdraws the prize. The time elapsed and gas used is recorded for each simulation for analysis.

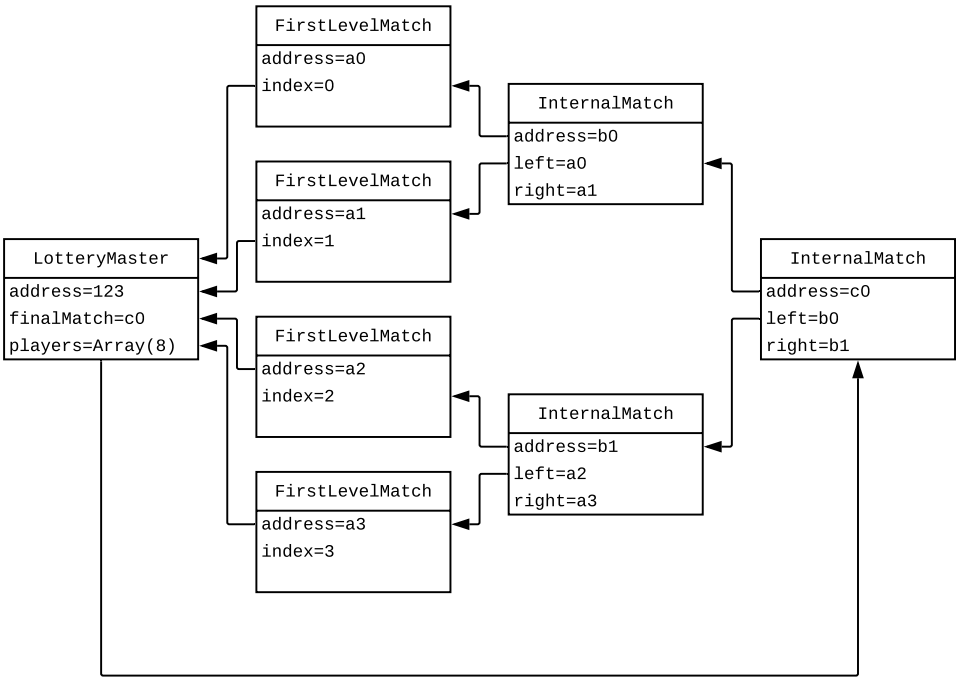


Figure 6: Contracts in a lottery of 8 players

4 Results

This chapter will present the results from simulating the lottery being set up and played. We primarily measure gas usage, which is the same on the local blockchain used for simulations as the main live Ethereum blockchain used in production. The transaction costs on Ethereum are measured in each transaction's gas usage. The transaction costs, which will be used interchangeably with gas costs, is the product of the gas usage and a gas price. The gas usage is a unitless number, while the gas price is denominated in *wei*, which is the most granular unit of ether. $1\ wei = 1\text{e}{-18}\ ether$. It's common to quote the gas price in gwei, which is simply a gigawei ($1\text{e}9\ wei$). When quoting prices in USD we operate with an ether price of 176 USD [1]. Unless otherwise stated, a gas price of 1.5 gwei is used in calculations. This is a quite low gas price, but usually high enough to have transactions included in the blockchain within 30 minutes or so.

In addition to presenting the results from simulation, we will analyze the security and scalability of the lottery. Some calculations will be made where both the gas usage results from the simulations and reasonable assumptions of gas price and maximum lottery prize will be used. The last section of the chapter will discuss the consequences of our design choices and trade-offs for the properties of the lottery, and suggest some alternative designs that will result in a lottery with different properties.

4.1 Gas usage and transaction costs

Even though all instructions in the EVM have a fixed gas cost, the gas usage of a transaction will depend on what code is actually executed, which can vary due to loops, external state, function arguments, and conditional statements. The gas usage of the lottery was measured by running simulations that made all the transactions necessary to (i) set up the lottery, which the lottery organizer must do, and (ii) set up and play the lottery, which is the complete gas usage of the lottery paid by both the organizer and the players. The lottery can be played in different valid ways that will result in different amounts of gas used. For instance, if some players don't make commitment or reveal transactions in a match, that will reduce the overall gas usage, but the lottery will still conclude successfully. In our simulation of playing the lottery, all matches are played completely by all players, so that

[1] Price from https://etherscan.io/ at 2019-05-07.

the maximum amount of transactions is made. This means that the gas costs measured are the upper bound for gas usage for this lottery. Due to the default winner logic in the match contracts, the lower bound of gas usage is when no `reveal()` or `commit()` transactions are made, but all other transactions are made. It is the upper bound that will be used in all calculations and analyses in this chapter's subsequent sections.

We found some tiny variations in average gas usage between different runs of a simulation with the same parameters. We expected the gas usage to be equal between runs, but the variance, which can be found in Appendix B, is so tiny we did not account for it.

Table 3: Average gas usage from simulation.

transaction	single match	dual match	# tx
`LotteryMaster.deploy()`	1087189	1087125	1
`LotteryMaster.setFinalMatch()`	49282	49282	1
`LotteryMatch.deploy()`	2472237	-	N-1
`FirstLevelMatch.deploy()`	-	1693728	N/2
`InternalMatch.deploy()`	-	1697819	(N/2)-1
`LotteryMatch.initFirstLevelMatch()`	74700	-	N/2
`LotteryMatch.initInternalMatch()`	71034	-	(N/2)-1
`LotteryMaster.deposit()`	73925	73925	N
`LotteryMatch.commit()`	83548	-	2(N-1)
`FirstLevelMatch.commit()`	-	76611	N
`InternalMatch.commit()`	-	88675	N-2
`LotteryMatch.reveal()`	43035	-	2(N-1)
`FirstLevelMatch.reveal()`	-	42884	N
`InternalMatch.reveal()`	-	42884	N-2
`LotteryMaster.withdraw()`	39277	39177	1

Table 3 is a list of transactions made in a simulation with 256 participants. The leftmost column describes the transaction made, while the rightmost column is the number of transactions of this type needed to successfully play a lottery. The middle columns are the average gas usage for each transaction for two different lottery designs. In the first design, we use a single match contract for all types of matches whether they are first level or internal, which requires an extra initialization step. In the second design, we use two separate match contracts for first level matches and internal matches. It is the second design that is described in Chapter 3, while the first design is similar to that used in [22].
Table 4 and 5 show the gas usage of only setting up and setting and playing a full lottery, respectively, for the design with a single type of match contract. Table 6 and 7 show the gas usage for the design with two types of match contracts, also

Table 4: Organizer gas usage. Single match contract.

N	Gas	ETH	USD	Gas / N	ETH / N	USD / N	Δ Gas / N
32	80023408	0.120	21.13	2500732	0.003751	0.660	
64	161466448	0.242	42.63	2522913	0.003784	0.666	22182
128	324354704	0.487	85.63	2534021	0.003801	0.669	11108
256	650139920	0.975	171.64	2539609	0.003809	0.670	5588
512	1301701456	1.953	343.65	2542386	0.003814	0.671	2777

Table 5: Total gas usage. Single match contract.

N	Gas	ETH	USD	Gas / N	ETH / N	USD / N	Δ Gas / N
32	90297000	0.135	23.84	2821781	0.004233	0.745	
64	182203802	0.273	48.10	2846934	0.004270	0.752	25153
128	366020720	0.549	96.63	2859537	0.004289	0.755	12602
256	733661484	1.100	193.69	2865865	0.004299	0.757	6328
512	1468940796	2.203	387.80	2869025	0.004304	0.757	3160

showing for only setting up and setting up and playing a full lottery. The dual match design compared to the single match design uses 50% less gas to set up, and 42% less gas to set up and play. As we can see in Table 3, the savings come from deploying the match contracts. This is because we deploy less bytecode and declare fewer variables with the dual match design.

As was expected from inspecting the smart contracts, the gas usage increases linearly with the number of participants. Even though the gas per participant also increases by a minuscule amount as the number of participants grow, this amount goes towards zero with more participants. For simulating the entire lifecycle of a lottery, this can be explained by the `deposit()` function which keeps expanding a dynamic array as players join.

We will use the measurements for dual matches from Table 3 in calculations later in this chapter to analyze ticket pricing and scalability. By using the amount of transactions necessary of each type, we see that the gas usage for a lottery of N players can be expressed as for just setting up the lottery $\frac{3391547 \cdot N}{2} - 561412$, and $\frac{4041505 \cdot N}{2} - 785353$ as the upper bound, i.e. all matches are played completely, for setting up and playing the entire lottery, while the lower bound, i.e. all matches are forfeited, is $\frac{3539397 \cdot N}{2} - 522235$.

The bulk of the transaction costs of a lottery is from setting up all the contracts, which must be done by the lottery organizer. The setup cost is 84% of the total transaction costs for the 512 player simulation for dual match contracts, and would be even more if not all players completed their matches. We see that by far most gas is spent on deploying each individual match contract, which must be done $N - 1$

Table 6: Organizer gas usage. Two types of match contracts.

N	Gas	ETH	USD	Gas / N	ETH / N	USD / N	Δ Gas / N
32	53690344	0.081	14.17	1677823	0.002517	0.443	
64	107956984	0.162	28.50	1686828	0.002530	0.445	9005
128	216490200	0.325	57.15	1691330	0.002537	0.447	4502
256	433556696	0.650	114.46	1693581	0.002540	0.447	2251
512	867690392	1.302	229.07	1694708	0.002542	0.447	1127

Table 7: Total gas usage. Two types of match contracts.

N	Gas	ETH	USD	Gas / N	ETH / N	USD / N	Δ Gas / N
32	63896706	0.096	16.87	1996772	0.002995	0.527	
64	128557218	0.193	33.94	2008707	0.003013	0.530	11934
128	257880692	0.387	68.08	2014693	0.003022	0.532	5986
256	516523956	0.775	136.36	2017672	0.003027	0.533	2979
512	1033816570	1.551	272.93	2019173	0.003029	0.533	1501

times. This cost was significantly decreased by splitting the match contracts into two types, the `FirstLevelMatch` and `InternalMatch` contracts. This is because each variable and each bit of bytecode contributes to the gas usage when deploying contracts, and the match contracts in the single match design contain code that is not necessary. Decreasing the number of methods, decreasing the number and size of variables, and decreasing the number of instructions, i.e. roughly lines of code, will decrease the gas usage. Since each match contract is not interacted with much – at most four times, two commits and two reveals – it seems wasteful to spend so much gas on deploying them on the blockchain permanently. This is a well-known issue, and it is possible to deduplicate common behaviour by using patterns involving *library contracts* and *proxy contracts*. The article in [63] demonstrate that gas usage for deploying contracts can decrease as much as 50% by using these patterns. It could also be possible to do away with the dedicated match contracts completely, and handle everything in the master contract, but we leave that to future work or a practical implementation.

While the price of ether and by extension gas is known to fluctuate heavily in fiat currency terms, we choose to primarily consider the cost of running the lottery only in the context of ether and gas price, even though we will also mention the price in USD in some cases to provide context. The transaction costs depend entirely on gas usage and gas price, and can be made an arbitrarily low fraction of the ticket price if we can set the ticket price arbitrarily high. The gas price is also known for fluctuating somewhat in terms of ether, but it is simply a result of market demand for resources on the EVM.

4.2 Ticket price

Since setting up and playing the lottery requires transaction fees to be paid in ether, a lottery with low ticket prices relative to the transaction costs is not feasible. As we saw in Section 4.1, most of the transaction fees are paid by the lottery organizer. It is reasonable to assume that in a real implementation of the lottery, there would be support for the organizer to take a fee of the total prize – a *house edge*, as a way to cover the costs for organizing the lottery. The fee should be at least so large that the transaction costs for deploying the contracts are covered. Since there is a risk that not enough players will join the lottery, and the organizer can claim no fee, it should be larger than only the transaction costs.

For simplicity, we will in subsequent calculations assume that the organizer fee is equivalent to the transaction costs for setting up and playing the entire lottery. Even though the cost of playing the lottery is paid by participants and not the organizer, we use it in the calculations, as it is at least related to the transaction costs of the lottery. An alternative could be to speculate on the risk of the lottery not starting, and account for that in the organizer fee, but since it involves speculation, we prefer to use the concrete total transaction costs of the lottery.

4.2.1 Lower and upper bound on ticket prices

Most common casino games have a house edge of about 1-10% [64], while large lotteries often have a higher house edge of 40-50% [65]. We therefore consider the organizer taking a 10% fee of the prize reasonable for both the players and the organizer. Since the prize is simply the sum of all the deposits, i.e. the income from ticket sales, and the transaction cost per participant is fixed, there is a lower bound on the ticket price if a 10% fee is to cover the transaction costs.

By setting a ratio r of what the organizer will take of the prize, and a gas price gas_price, we can find a minimum ticket price $ticket\_price_{min}$ by using the gas usage gas_usage for 512 players from Table 7 for setting up the lottery per player. The expression is simply $ticket\_price_{min} = \frac{gas\_price \cdot gas\_usage}{r}$. If the ticket price is lower than this, the fee is not large enough to cover the lottery's transaction costs. In our design, with a 10% organizer fee and a 1.5 gwei gas price, this results in a minimum viable price of each ticket at about 0.03 ether, which at the time of writing is about 5 USD. That's a price we consider acceptable, even though the dollar price can change by at least an order of magnitude within months, as the ether price fluctuates a lot.

While the ticket price can be set arbitrarily high to make the necessary organizer fee arbitrarily low, it will be discussed in Section 4.4 that a high prize could be a security concern. If we assume a maximum acceptable prize as a security parameter, the ticket price will also have an upper bound $ticket\_price_{max}$. This bound will

be dependent on the organizer fee r and the number of participants N. The prize of the lottery is expressed as $prize = ticket\_price \cdot N(1-r)$. If the prize is bound to a maximum value for security reasons, it follows that the maximum acceptable ticket price will be $ticket\_price_{max} = \frac{prize_{max}}{N(1-r)}$.

The upper bound on ticket price is dependent on both the amount of participants and the organizer fee. If we assume that the organizer fee will not be higher than about 50%, we see that a maximum acceptable prize entails a maximum amount of participants, and is hence a scalability issue. The implications of this will be discussed further in Section 4.5.

4.2.2 A ticket price of zero

Even though we think of our lottery as a gambling game where participants pay a small ticket price in order to compete for a large prize, the protocol of our lottery could be used as a more general leader election scheme. Doing this, we essentially remove the monetary incentives of each player to select a winner, and exchange it with incentives to select a leader who gains some responsibility. A simple variant of the lottery as a leader election scheme would just remove the cost of tickets and the prize, as the conditions for participation is something else than paying for a ticket, and the winner is not paid a share of the ticket fund.

If we assume that participants are willing to pay for the transaction costs of playing, a lottery with a ticket price of zero would technically work exactly as one with money involved. However, an important property of our lottery with money incentives is that there is no advantage in colluding. It is the case that any single player has a $\frac{1}{N}$ probability of winning, and any collusion of n players has a $\frac{n}{N}$ probability of one of its players winning. However, a collusion will have a better possibility to predict the result than any single player. If two players of a collusion are to play against each other in a match, they can choose secrets so that they can tactically choose who should win. If the result of the final match is used for something external, then two colluding players in the final match could arbitrarily choose the outcome and who wins the tournament. Finally, if the lottery is used as a leader election scheme, the possibility of a collusion being able to predict the outcome – which could be the case in our protocol – is usually not desirable. This is not the case in another leader election scheme such as RanDAO [42], where a collusion of $N-1$ players are no better able to predict the outcome than any single player.

Another important property of our lottery is that it is pari-mutuel, meaning the prize is not higher than the sum of all ticket purchases. If the ticket price were zero, this would not be the case, which would incentivize sybil behaviour, as controlling all participants would be advantageous if the expected value of participation is

higher than zero [33]. This does not necessarily make the lottery protocol itself bad at a not pari-mutuel lottery, but the conditions for participation might have to account for sybil resistance in a different way than charging a price for a ticket.

4.3 Security

Most applications on a blockchain will more or less inherit the blockchain's own security model. There is always a risk of censorship, loss of connectivity, block reorganization, and more, but open blockchains could still be the most secure computing platform we have for MPC without a trusted intermediary. A lottery is special in that the prize can be incredibly high, and for that reason it might attract more motivated attackers than applications that do not handle large amounts of value in a single transaction. By viewing security from an economics perspective, where an attack has a cost c, a chance of success p, and an exposed value v that the attacker will gain if successful. If the expected reward $c \cdot v$ is so high that $c < v \cdot p$, the risks of an attack are too high. The profit potential for successful attack on a large lottery can be so high that extraordinary measures might be taken by adversaries to succeed. We must therefore review the most common security concerns such as censorship, network attacks, and block reorganizations when discussing the security of a lottery.

4.3.1 Loss of connectivity

The lottery is inherently interactive in that participants need to make commitments and reveal secrets during the lottery playing phase. Since players who don't make a commitment or reveal within the time limits will lose, loss of connectivity is an issue. Since the lottery protocol requires interactivity, there is little we can do to mitigate this other than using longer time intervals between the steps (t_{commit}, t_{reveal}, t_{play}) in the match contracts, so that players have a chance to reconnect before a time limit is reached.

While players could mitigate against losing connectivity by outsourcing the interaction with the lottery to some third party service, that would entail sharing of secrets with a third party, which is a security consideration of its own. Players could run the lottery from servers under their control in data centers at different geographical locations than their web browser, and thus be quite safe against losing connectivity with minimal risk of leaking the secrets, but this is complicated for many users.

4.3.2 Blockchain reorganizations

A block reorganization attack is a concern in that the results of a lottery can be reversed if a corrupt powerful miner is not happy with the result. Reversing the entire lottery, however, is probably not possible as block reorganization attacks are

very expensive and difficult to perform on long subchains. A cheaper attack would be to wait for one's opponent in a match to reveal their secret, and then reorganize the blockchain if the result is not favorable. While it's certainly difficult to do so, we don't know exactly what the expected reward and cost would be for such an attack if the lottery prize is extremely high.

This concern can be mitigated by using longer time intervals between steps in the matches, as block reorganization attacks get more expensive the more blocks are involved. The production of blocks is quite inherent to the blockchain platform itself, and there is not much an application on the blockchain can do about that. One way of decreasing the risk of a block reorganization happening is using a blockchain that has a high mining power and much decentralization among miners, so that coordinating an attack is harder.

4.3.3 Censorship and transaction blocking

Network attacks

While blocking of the network and eclipse attacks are relevant for all network applications, it is commonly assumed that long lasting attacks of this type will not happen. Even though there's always a risk of it happening, we can choose security parameters that minimize the consequences of such attacks. Again, with an interactive lottery, we can't do much more than to increase the time intervals between time limits in matches, so that the chance of getting one message through during that interval is high even when under attack. For targeted attacks on the network, players can also hide their location by accessing the network with privacy enhancing tools. Due to general network attacks being quite peripheral to the topic of this thesis, they are not discussed in detail, and the blockchain is assumed to be fairly resistant to these kinds of attacks.

DoS attacks

A denial-of-service attack (DoS) is done by flooding a network with bogus messages so that it is incapable of responding to legitimate messages. This can be done on a blockchain by broadcasting a large number of transactions with high transaction fees, so that miners will only include the bogus transactions in the blocks and ignore other transactions with normal transaction fees. Such an attack is costly to withhold over time, as transaction fees must be paid. But if the expected value of successfully blocking an opponent in a match in a lottery is high, it can very well be worth it. Such a transaction flooding attack can easily be countered by broadcasting a transaction with even higher transaction fees. This fact makes the relationship between attacker and defender asymmetric, as the defender only needs *one* transaction to go through, while the attacker must block *all* other transactions. A lottery client should take this into account and be ready to make transactions with high

transaction fees if it suspects a DoS attack is under way.

Censorship

For a blockchain to have a high degree of liveness, i.e. transactions will be recorded rather quickly and reliably, we must assume that miners will include transactions by no other discrimination than the size of the transaction fees. However, miners are free to choose which transactions they include in the blocks they produce. They could for any reason refuse to include a transaction from or to a certain address, including from certain participants in a lottery. The main concern is an adversary paying miners bribes on the condition that transactions from certain participants are not included, or that miners themselves play in the lottery with the intent of abusing their power to censor. We see from how the tournament tree looks like that one would only need to block transactions from $log_2(N)$ participants to make sure that one wins each match by the other player failing to make a commit or reveal transaction.

While it's likely that a non-corrupted miner will include transactions censored by other miners, and that the likelihood of that happening increases with time, corrupted miners can also choose to ignore blocks that are produced by honest miners, i.e. the selfish mining strategy. Such a situation could have the corrupted miner cause a block reorganization where blocks including censored transactions are replaced by the corrupt miner's blocks. Such a combination of censorship and block reorganization can be quite powerful if the corrupt miner or collusion of miners controls a large share of the mining power.

While a combined censorship and block reorganization attack backed by a large portion of the mining power is very hard to mitigate, less powerful censorship attacks can be mitigated in several ways. One is again to make sure the steps in the match contracts are sufficiently long. Another is to make the lottery less interactive and somehow reduce the amount of transactions that are necessary for completion. Another is to use anonymous transactions in which miners cannot know what the result of the transaction will be at mining time, but this cannot be accomplished without major changes to either the lottery design or the way mining on the Ethereum blockchain works. The issue of censorship is discussed in more detail in Section 4.4.

4.3.4 Compromised client and phishing

Compromised secrets

If secrets are generated on a client such as a web browser, an adversary could trick participants to using a compromised client that leaks secrets to them. While this security consideration is quite broad, as an attack would go through a player's computer and there is little we can do about a compromised computer when designing

the lottery. Following standard practices for designing secure applications can be done in the application layer of the dApp.

Compromised client

Since it is up to players to verify that the lottery is set up correctly by validating all the smart contracts, a compromised client could falsely validate a lottery that is set up in an adversary's favor. We assume that each player is capable of verifying that the lottery and tournament is set up correctly. An adversary could make it look like the lottery is set up correctly by luring players into a fake site with phishing. Verifying that the lottery is set up correctly is done by the client, as the smart contracts cannot do this on their own in our current design. E.g. the reference to the final match contract is set in the master contract after initialization. The master contract does no more validation than verifying that the final match contract is a an Ethereum address. A malicious lottery organizer could make a bogus final match contract in which they are guaranteed to win, and which is not related to the actual tournament at all. Even if the final match is unfair, unknowing players can make deposits to the master contract, and the master contract will pay the prize to whoever is the winner of the match contract it was set up to use.

4.4 Cost of a censorship attack

The most concerning attack is probably one where miners tactically censor the transactions of one participant's opponents through the entire lottery. If a participant is not able to make their commit and reveal transactions, they lose that match. From the miners' perspective, a block reorganization attack is risky and expensive as it might fail, but a censorship attack is almost cost-less for a powerful collusion of miners, as failing does not necessarily have a cost. This means that if a miner is completely unconcerned with keeping the network healthy, even a marginal bribe would make it worth to censor someone. One could counter a small bribe by increasing the gas price in a transaction, so that the miner will be paid more in transaction fees than what the bribe is worth, but a briber would still have an advantage, as the rewards of a successful attack would fund their activity.

In practice, not many miners are willing to take bribes for censorship, as they are either idealistic or have an interest in maintaining the reputation of the blockchain as a secure computing platform. At least, the latter point is the case if miners have skin in the game of the blockchain, such as owning mining hardware specific for that chain or having large holdings of the currency of that blockchain. If a blockchain gets a reputation for being susceptible to censorship, the demand of the chain's cryptocurrency might decrease significantly, and thus indirectly hurt miners who engage in censorship. Nonetheless, it's still useful to assess the feasibility of a censorship attack, as the value at stake in a large lottery can be extraordinarily

high.

A lottery has L levels in its tournament and $N = 2^L$ players and a prize of $pr2^L$ where r is the ratio of ticket deposits that go to the prize and p is the ticket price. Ideally, each player has a $\frac{1}{2^L}$ chance of winning, so that the expected value of participation is $\frac{pr2^L}{2^L} = pr$. For each round an adversary can successfully censor their opponent, they double their chance of winning and thus double their expected value. Note that the expected value doubles for each round, so that in the first round the expected return is pr, while in the last round it's $2^{L-1}pr$. So if bribing miners to censor one's opponent is barely worth it in one round, it's certainly worth it in the next round.

A censorship attack has a probability of succeeding for each block produced. If we assume there is not a powerful selfish miner that has the power to both censor transactions and force the network to abandon blocks that include transactions targeted for censorship, there is a probability that an honest miner will include the targeted transactions. Since a coalition of selfish miners can be effective with as little as 25% of the mining power, and certainly at 50% of the mining power, we assume that in the worst case only 50% of the miners are honest. If the fraction of honest miners is any less than that, then censorship by selfish miners dominates the threat model. We see that even with just half of the miners being honest, the likelihood of a transaction being included in a block gets very high after just a dozen blocks. $p_{included} = 1 - 0.5^{12} = 99.9756\%$. Therefore, the time limits in matches should be sufficiently large so that the risk of a successful censorship attack without the selfish miner strategy is negligible.

We can measure the reward of censoring a participant in one round by the increase in expected return for the adversary. As noted before, this reward will increase exponentially as we get closer to the final match in the tournament. This also means that the reward in large lotteries will be correspondingly large. If we assume a dollar value of a ticket be to 10 USD and we have $2^{16} = 65536$ participants, the reward of censoring your opponent in the last match will be $\frac{10 \cdot 2^{16}}{2} = 327680$ USD, which at today's ether price makes it well worth it for a miner to risk losing some block rewards in order to get a chunk of the prize [2].

As the adversary's opponent in a match reveal their secret by broadcasting a reveal transaction to the network, the adversary and the colluding miners, or anyone with knowledge of the adversary's secret, will know what the result of the match will be after the reveal transaction is broadcast. Since the match consists of a digital coin toss, in 50% of the cases there will be no need for censorship at all for the adversary to win. The adversary can choose to censor only when they will not win honestly. This puts the honest participant at a disadvantage, as even though they

[2] The block reward in Ethereum is as of 2019-05-07 2 ether or 352 USD (https://etherscan.io/).

can counter the bribe by making a transaction with a high gas price, they won't know whether they will win or not until the adversary has made their transaction.

This means that in the case of an honest player playing a match against an adversary engaging in censorship, the adversary would be willing to spend double the amount to bribe miners of what the honest player is willing to pay in transaction costs to counter such a bribe. An honest participant would only be willing to spend the entire expected value at that level, while the adversary is willing to spend double that if they know they would lose by playing honestly.

As to whether miners would accept bribes to censor, we don't know of any estimates of how large a bribe must be for miners to collude in such a way. Assuming it is possible to bribe a powerful miner to censor transactions at all, the possible rewards of doing so increases as the lottery gets larger, so it will eventually be high enough. An opportunistic collusion of miners could also perform such an attack on their own by participating in the lottery – removing the necessity of bribing.

Based on the analysis in this section, we conclude that our proposed lottery is only secure under the assumption that there is no successful selfish miner behaviour on the blockchain. This is because a selfish miner collusion with about a third of the mining power can in some cases effectively censor transactions, and our lottery requires transactions to be included in blocks within reasonable time. A selfish miner collusion with less than 25% of the mining power is very unlikely to succeed, while the likelihood of success increases the higher their fraction of mining power is. If such an attack can happen in practice, participants in the lottery ultimately have to trust the miner collusion to act honestly, and introducing a trusted intermediary goes against the entire purpose of distributed lotteries.

This can only be partly mitigated by having more time between time limits in matches. Such a censorship attack is essentially costless if we ignore indirect costs such as the blockchain's reputation being hurt, and can be sustained indefinitely as long as the collusion of miners is powerful enough. We instead would recommend to not hold large lotteries or lotteries with very high prizes, and to be vary if the blockchain has powerful miners or high risk of miner collusion.

4.5 Scalability

It's common for lotteries to have millions of participants. Their large scale is an important characteristic, so it's natural to explore the scalability of our implementation. While the Ethereum platform has no problem with having any amount of connections to participants all over the world, it has a limited amount of transactions that can be processed per time unit. Even if transaction costs can be overcome by making them low relative to ticket prices, the limit on transaction throughput will stay the same. The stakes involved in the lottery will also get higher the more

participants join. This fact might create incentives for miners to censor transactions as described in Section 4.4. Both the limits on transaction throughput and the security exposure of a higher prize will put a limit on the scalability of the lottery. The former is a consequence of the limited resources on a global blockchain, while the latter is due to the possibility of an attack, and is thus a matter of security. The scalability of the lottery will be analyzed from both perspectives below.

4.5.1 Transaction throughput

Our lottery has clearly defined limits for the amount of transactions that need to be made for each participant. The amount of transactions of each type and its average gas usage is listed in Table 3. Setting up the lottery requires $N + 1$ transactions, each participant joining takes one transaction for a total of N, playing each of the $N - 1$ matches takes at most 4 transactions each, and the winner needs a single transaction to withdraw the prize. So the maximum amount of transactions for a successful lottery will be $6N - 2$.

The time interval in which the transactions need to be made is important, as the intensity of transaction demand will vary during the lifecycle of the lottery. We expect a ticket purchasing period that lasts for several days where the demand for transactions will not be high per time unit. Right after the deposit phase, when the matches on the first level can be played, there are $\frac{N}{2}$ matches that all need to be interacted with at the same time. If the time interval between match phases of these matches is too low, the blockchain will not be able to handle all the transactions that need to be made. The demand for transactions after the first match will exponentially decrease for each level as the amount of matches is halved for each level in the tournament tree.

The amount of participants is 2^L and there are 2^{l-1} matches for each level if we count l from L at the first level matches to 1 for the final match. The commit step and reveal step for each match require two transactions each. The amount of transactions needed for each step at each level will then be $txs\_step_l = 2 \cdot 2^{l-1} = 2^l$ – one for each remaining player. Ethereum has a transaction throughput capacity of roughly some amount of transactions per block tpb. The space for transactions in an Ethereum block is actually limited by the sum of gas used by its transactions. The commit transactions needed to play our lottery use on average approximately 80000 gas per transaction, which is the number we will operate with. Since blocks are mined on average at a fixed rate, transactions per block is equivalent to transactions per time unit. Each step in a match lasts for a number of blocks td which is set when each match contract is deployed. For it to be theoretically possible to perform all the transactions for each match, td must be set so that $td > \frac{2^l}{tpb}$.

Ethereum currently has a capacity for about 100 transactions of our type per

block [3]. Using this value for tpb, we can chart some values of what td ought to be in the first level matches for various amounts of participants.

Table 8: Estimates of td if all transactions on the blockchain are used for our lottery.

L	N	td	td in s	td in m	td in h
8	256	3	45	0.8	0.01
10	1024	11	165	2.8	0.05
12	4096	41	615	10.3	0.17
14	16384	164	2460	41.0	0.68
16	65536	656	9840	164.0	2.73
17	131072	1311	19665	327.8	5.46
18	262144	2622	39330	655.5	10.93
20	1048576	10486	157290	2621.5	43.69

Table 9: Estimates of td if 10% of the transactions on the blockchain are used for our lottery.

L	N	td	td in s	td in m	td in h
8	256	26	390	6.5	0.11
10	1024	103	1545	25.8	0.43
12	4096	410	6150	102.5	1.71
14	16384	1639	24585	409.8	6.83
16	65536	6554	98310	1638.5	27.31
17	131072	13108	196620	3277.0	54.62
18	262144	26215	393225	6553.8	109.23
20	1048576	104858	1572870	26214.5	436.91

The most realistic scenario is that only a small fraction of all the transactions in a block are used for the lottery. We also see that for a lottery of any size, there will probably be a quite high demand for transactions, which will in turn raise gas prices. This has implications for what the minimum viable ticket price should be, as it cannot be assumed that the gas price will be low for the first level of the lottery unless td is very high. We that the bulk of transaction costs comes from deploying the lottery contracts, so thee increased gas price during the playing phase will have a limited impact on the total transaction costs.

Although td can be set arbitrarily high, we probably don't want the lottery to drag on for weeks before a winner is determined. It seems the lottery will face scalability issues in the order of 100000s or from 2^{17} participants if it is to be completed within days. The number of transactions needed will decrease exponentially with higher levels, so the first two levels will account for 75% of the total time of the playing phase.

[3] As of 2019-05-07, the gas limit is 8000000 gas per block (https://etherscan.io/).

Table 10: Time for the entire lottery if 10% of the transactions on the blockchain are used for our lottery.

L	N	total time	time in s	time in m	time in h
8	256	110	1650	27.5	0.46
10	1024	420	6300	105.0	1.75
12	4096	1650	24750	412.5	6.88
14	16384	6568	98520	1642.0	27.37
16	65536	26230	393450	6557.5	109.29
17	131072	52446	786690	13111.5	218.53
18	262144	104876	1573140	26219.0	436.98
20	1048576	419450	6291750	104862.5	1 747.71

There are plans for Ethereum to increase the number of transactions it can handle by implementing proof-of-stake (PoS) and sharding. While this might make our lottery capable of handling more participants, we don't know whether such a change in the protocol might introduce a new threat model that makes the lottery less scalable in other ways.

4.5.2 Transaction costs and max prize

Due to the size of the prize increasing linearly with the number of participants, a censorship attack by miners gets more profitable the more participants there are, which puts a limit on scalability. Even if a censorship attack is unlikely to succeed, it cannot be ignored when it can give high rewards to dishonest miners. Since the lottery prize is decided by the ticket price as well as the number of participants, the ticket price can be set so low that the total prize is not high enough to encourage manipulation by miners. The ticket price, however, is bound to a minimum value where the transaction costs for playing the lottery get prohibitively high compared to the ticket cost. Using a gas price of 1.5 gwei and the gas usage for setting up and playing a dual match lottery found in Section 4.1, we will present various charts to illustrate the scalability of our implementation of the lottery.

The *maximum prize* is a security parameter used in this analysis. A higher maximum prize exposes more value to be lost in case of a successful attack, and so the security risk increases with a higher maximum prize. The lottery organizer and participants can judge what their tolerance for a max prize is, and we will operate with values up to 3000 ether for illustration. Another important variable is the *transaction cost ratio*. This is the ratio o f ticket price to transaction costs for setting up and playing a lottery, i.e. $r = \frac{gas\_usage \cdot gas\_price}{ticket\_price}$. The prize is also entirely dependent on the ticket price, as the calculations for simplicity's sake assume the organizer takes a fee equal to the transaction cost ratio times the sum of all deposits, i.e. $prize = ticket\_price \cdot N(1 - r)$ and $fee = ticket\_price \cdot Nr$ where N is

the number of participants. The ticket price is implicitly used in the charts, as any transaction cost ratio is mapped bijectively to a price when the transaction costs are fixed.

Figure 7 shows the ratio of the ticket price going to transaction costs as a function of number of participants and a max tolerable prize. Figure 8 show the prize for the lottery for a given number of participants and ratio of transactions costs to ticket price. The charts show us the maximum scale of the lottery under different conditions for maximum tolerable prize and ticket price going to transaction costs. With the assumptions that the transaction costs can be as much as 20% of the ticket price, and the maximum prize being about 1500 ether, this scalability limit in terms of number of participants approaches that of the analysis of transaction throughput.

Figure 9 shows the transaction cost to ticket price ratio on the left y-axis which the thick blue line plots. The right y-axis shows the total prize for a given ticket price and is plotted by multiple thin lines that each represent a specific number of participants. The chart gives some idea of which tickets prices can be used for different amounts of participants and different maximum prizes. For instance, if a cost ratio of 0.1 is required, the thick blue line indicates that the ticket price will be 0.03 ETH. At that price, only lotteries with less than $32768 = 2^{15}$ will have a prize of less than 1000 ETH. If we start from the other direction and consider a lottery with $8192 = 2^{13}$ participants, plotted by the thin pink line, it can handle ticket prices of slightly more than 0.1 ETH if the prize is to be less than 1000 ETH, and a ticket price of 0.06 if the prize is to be below 500 ETH. Furthermore, it shows that lotteries with few participants can handle quite high ticket prices without risking too much exposure by a high prize, while lotteries with more than 2^{16} participants can barely handle ticket prices so low that the cost ratio is nearly prohibitive unless a very large prize is acceptable.

4.5.3 Scalability limits

It's difficult to say anything authoritative on the scalability in terms of maximum prize, as it is unknown how high a sensible maximum prize should be. If a large prize and organizer fee is acceptable, then the main scalability bottleneck will be in the transaction throughput of the Ethereum blockchain. If the transaction throughput were to be increased, or the number of necessary transactions to complete the lottery can be decreased, the bottleneck could be in the maximum acceptable prize. The discoveries in this section allows a lottery organizer to make informed choices on setting a ticket price, fee, and amount of participants, but do not help in figuring out how high a sensible max prize should be. We also see that the interactivity of the lottery design has consequences for the scalability in that it requires many

transactions.

If the lottery playing phase is to be concluded with days and not weeks, and we assume that 10% of all transactions on Ethereum can be related to the lottery for a while, the scalability limit of the current design is $2^{17} = 131072$ participants. If we only consider ticket prices where the ticket price to transaction costs ratio is less than 0.2, 2^{17} participants is acceptable from a maximum prize perspective if the max prize is at least 1587 ether or 279312 USD.

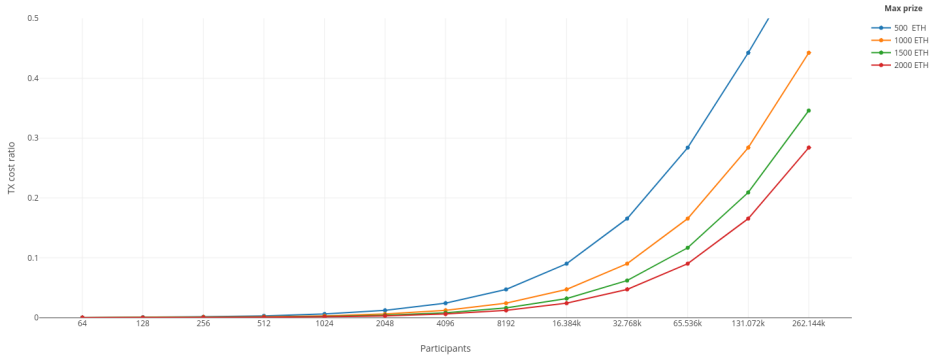


Figure 7: Cost ratio as a function of participants and max prize.

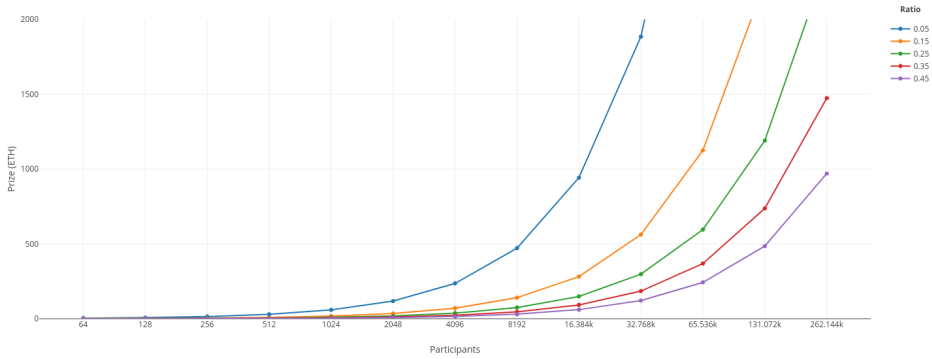


Figure 8: Prize as a function of participants and cost ratio.

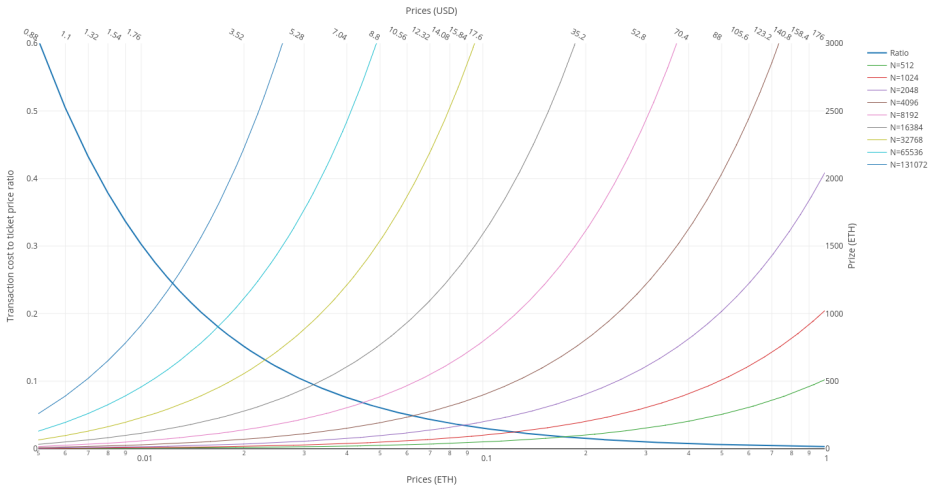


Figure 9: Cost ratio and prize as a function of participants and ticket price.

4.6 Analysis

4.6.1 Consequences of interactivity

Based on the implementation choices of the lottery and the above results and discussion, it is possible to get an insight into important properties and trade-offs of the lottery. The lottery is interactive in that participants have to make several transactions in order to complete it successfully. The interactivity is a trade-off made in this design as opposed to the other blockchain lotteries of [20, 21], where fewer transactions are necessary, but upfront deposits are needed from all participants. We found that in our design, this has consequences for the scalability for several reasons. One is that the amount of transactions needed to play the lottery will be so high that it will start to congest the entire Ethereum network. Another reason is that the transaction costs set a lower bound on the minimum viable ticket price. As we operate with a maximum lottery prize for security reasons, this means that lotteries with higher ticket prices will have less capacity for participants.

What is gained by the interactivity of the scheme is that the lottery is guaranteed to complete. Deposits are not needed as it seems a player has nothing to gain by not following the rules. A collusion of participants will not be able to predict the outcome of the randomness other than the outcome of the matches between players in the collusion. Due to secrets being committed to and not revealed until both players of a match have finalized their commitments, a collusion will not have any advantage over a single honest player.

We found that it is the cost of setting up the lottery by deploying contracts that is the most significant transaction cost. Since this cost directly influences the minimum viable ticket price, reducing the transaction cost of setting up the lottery would make the lottery capable of having more participants. It is likely that reducing the transaction costs can be done by using different design patterns in the smart contracts. Simply optimizing the length of the variables may also save some transaction costs. The current implementation only uses `uint256` types for numbers, but most of the numbers will never be so big that they need 256 bits to represent them. However, reducing variable length by itself won't save gas, as the EVM operate with a 32-byte (256 bits) word size. Gas could be saved by packing variables of lesser length into structs of 32 bytes, but we leave this optimization to future work.

The number of transactions needed to play the lottery has consequences for both the scalability of the lottery and the total transaction cost. This number can be reduced by playing the digital coin tosses *off-chain*. Off-chain means that some part of a protocol is handled between the players over another communication channel than directly on the blockchain. Consider the digital coin toss after the commitments are made. As the outcome at that point is determined if both players reveal their secrets, it's not actually necessary to do that on the blockchain. The players could simply reveal their secrets to each other in private, and then one player would not need to make an actual transaction, as the losing party would not gain anything but still pay a transaction fee. If only the winner makes a reveal transaction, they will win regardless of what the loser does, as not making a reveal transaction is interpreted as forfeiture. Doing this requires no change in the smart contracts from what they are in the current implementation.

It is possible that this idea of negotiating the matches off-chain could be taken even further by using hierarchical deterministic secrets (HDS) [66]. The idea is that participants make a single commitment to a public key at the beginning of the tournament. In each match of level i, players derive a private key of index i from the committed public key. This private key will serve as the secret in a normal digital coin toss protocol. The way HDS work is that a parent asymmetric key pair (sk_p, pk_p) can generate deterministic child key pairs with specific indices. By knowing just the public parent key, one can generate child public keys $\{pk_0, ..., pk_i\}$, and by knowing the private parent key, one can generate child private keys $\{sk_0, ..., sk_i\}$ which correspond to public keys with the same index. This means that a single public key can serve as a series of commitments by using its child public keys. The secrets will be verifiable as it is possible to verify that a private key corresponds to a public key if one knows both. Using this idea, a lottery could be negotiated off-chain once all players have made their one commitment to a parent public key.

Since it must be necessary to enforce the rules in case participants do not engage in the off-chain negotiation, all the contracts in the tournament would still have to exist, but would not necessarily be used. If commitments are made during the purchasing phase, the spike in transaction demand when playing the first matches could be drastically reduced, hence improving the scalability of the lottery. But implementing this would require significant changes from our current implementation.

4.6.2 Tournament without a full binary tree

The design of the lottery uses a tournament that is required to be a full binary tree which must be set up before players join. This limits the amounts of players in the lottery to those that is a power of two. While this limitation might make the lottery impractical, it makes it easier to reason about its properties in theory. If one were to allow for more flexible amounts of participants by allowing a non-full binary tree in the tournament, one would have to make sure that players would still have the same probability of winning, or at least the same expected outcome. For instance, if one handles a tournament with $2^L + 1$ players by having one player in a separate subtree, that player would advance to the final match without playing a single match. Should that player then be eligible for a smaller prize, or pay a higher ticket price?

One option to handle lotteries with any amount of participants is to make the prize proportional to the number of matches the winner has won. Say, if there are 12 participants, so that the tournament is a not complete binary tree of height 4. The first 8 players form a complete binary tree of height 3 – the left subtree, while the last 4 participants form a not complete binary tree of height 2 – the right subtree. The final match has the roots of the two subtrees as children. If a player from the left subtree wins the tournament, they will get the entire prize. If a player from the right subtree wins, they will only get half the prize, as they played one match less. This design raises the question of what should be done with the remainder of the prize if a player from the right subtree wins. It could be given to the other player in the final match. This would keep the essential property of each participant having equal expected value from participating, but participants in the right subtree will not have the opportunity to win the complete prize. If instead of 12 players, there are e.g. $2^4 + 1 = 17$ players, the only player in the right subtree would be guaranteed to enter the final match, but will only play one match, and thus only be entitled to $\frac{2}{17}$ of the prize. Whether this design is sound and whether it has any non-obvious vulnerabilities would be interesting to explore.

Another possible solution is to have some matches consisting of three players instead of just two. Unfortunately, due to the time limits, this can give an advantage

to two players in the same match colluding. In a match with three players, each player would ideally have a $\frac{1}{3}$ probability of winning, so that in a match with two colluding players, they would have a $\frac{2}{3}$ probability of winning. But if the secret of the non-colluding party Alice is revealed first, the colluding parties Colin and Lucy have three options. (i) Only Colin reveals, (ii) only Lucy reveals, and (iii) both reveal. Assuming each player who reveal has an equal probability of winning and that their secrets are uniformly random, the first two options each have a $\frac{1}{2}$ probability of either Colin or Lucy winning, while in the third option there's a $\frac{2}{3}$ chance of either winning. The slim probability that Alice wins in all the three options is just $\frac{1}{2 \cdot 2 \cdot 3} = \frac{1}{12}$, making the either Carol or Lucy win in 11 of 12 cases.

It is possible that there is a design in which a tournament with any amount of participants can be played fairly, but it would involve some considerable design changes from our lottery that is beyond the scope of this thesis.

4.6.3 Mitigating a censorship attack

A possible vulnerability in censorship of transactions was discovered. Even though the risk of a censorship attack is unknown, if we assume the possibility of a miner or collusion of miners with a majority hashing power, it would at some size of the lottery prize be in their economic interest to launch such an attack. A censorship attack is possible because of the time limits of matches, because a player unable to make transactions will lose. The time limits are however necessary to prevent a single participant halting the entire process.

A possible way of mitigating a censorship attack is to annul the result of the tournament if the winner won a certain fraction of their matches by forfeiture. This could possibly do some collateral damage in that an honest winner could risk being suspected of cheating. It could also make the off-chain negotiation mentioned earlier in this section impractical.

However, increasing censorship resistance is a too broad topic to be in the scope of this thesis. We operate with the common assumption that a blockchain system such as Ethereum is censorship resistant to a certain degree, and we can only caution lottery organizers and participants to judge the risk by the size of the lottery prize.

5 Discussion

5.1 Programming tools

We were generally able to use existing tools to our advantage for development and simulation. Truffle was used to compile and manage smart contracts. We used Truffle's testing environment and smart contract abstraction to successfully run unit tests and simulations on our code. Ganache is a program that starts an Ethereum RPC server and blockchain on your local computer. This allows one to quickly deploy contracts, make transactions, mock behaviour, and measure performance without using something out of the developer's control such as the main Ethereum network or even a global testnet.

By limiting ourselves to using the smart contract abstraction that comes with Truffle, we were not able to overcome some shortcomings when simulating and testing. Truffle's smart contract abstraction is designed to be used in web clients of dApps that interact with the Ethereum blockchain. Since each interaction with the blockchain involves an RPC, such interactions are handled with JavaScript promises – a language feature that facilitates making asynchronous applications. For every transaction RPC that is made, Truffle will wait until it is confirmed in the blockchain before it performs the next RPC. This limits how we can use Truffle when simulating, as a simulation may involve thousands of transactions, many of which can be made in parallel. In our code, we have to wait for each transaction to be confirmed before the next one is made, even though waiting for confirmations should in many cases ideally be done in parallel. As our simulations require some transactions to be made in specific order, such as making commitments before revealing secrets, we have to use promises in some cases. We could avoid this limitation by making all our transaction RPCs by using web3js directly, but that would involve a more complicated simulation program. This limited out simulations to use no more than 512 participants, as using more would take a long time. However, we don't think simulations with more participants would provide much more useful data.

Our smart contracts are highly dependent on timing conditions. The time limits are specified in fields that denominate block height. E.g. a match contract is initialized with three limits: `tCommit`, `tReveal`, and `tPlay`. These are essential for security in a real deployment, but testing and simulating with many time conditions can be hard. In our testing environment we run a local blockchain using Ganache.

Due to the issue with Truffle discussed above, we have to enable feature called *auto mining* in Ganache, so that a new block is generated with each transaction. This is not how the blockchain would behave in a live situation, so instead of altering our smart contract code to fit into our testing environment, we chose to not use time limits in the contracts when testing and simulating. We could configure a much more complicated testing environment where we can control which transactions are mined at which block heights etc., but chose to not spend time doing it as we were nonetheless able to run useful simulations and tests.

5.2 Blockchain security

The common security assumptions, attack vectors, and threat models of the underlying blockchain can be essential to understand and consider when deploying applications on top of the blockchain. Blockchain security, however, is a complex and not well understood topic. Researchers and practitioners in the space have identified various attack scenarios and concerns related to the security of a blockchain. We have discussed some of these, such as selfish mining, censorship, and block reorganization. We consider these issues important to discuss when developing an application on top of a blockchain, but since we lack real data on what happens during such an attack, as it has not happened on a major blockchain, discussing it is somewhat speculative.

Our analysis on the security of our implementation is based on uncertain assumptions and speculative attack scenarios. While something like censorship of transactions by miners certainly is possible, there is no research indicating that it is common or even in rational miners' interest to do so. On the other hand, blockchain platforms for smart contracts is a relatively new phenomenon, so it may well be a latent threat that will be a more real concern in the future. While we think that discussion on theoretical attacks is useful, we admit that the results from such discussion cannot give a concrete results that are applicable in all cases. However, when designing protocols that handle large amounts of money, a prudent approach should be taken as it's better to fail on the side of being too cautious rather than too reckless. With that in mind, our analysis on potential attacks is indeed useful.

5.3 Experimentation

Experimentation within a real setting is something we were not able to do in this project. The plan was to limit the scope to only make a proof-of-concept implementation, but the lack of experience from a real deployment makes it difficult to answer whether Ethereum is a good platform for a distributed lottery. While some properties of the blockchain and applications built on top of it can be analyzed by merely using theory and assumptions, the live version of a blockchain like

Ethereum is a complex system with many stakeholders and participants, many of whom are opportunistic or right out malicious. One infamous example of something unexpected going wrong is the DAO and reentrancy bug exploit which resulted in a hardfork of Ethereum [67].

Not only is a dApp likely to face unexpected issues when deployed on a live network, but users of the dApp might have various expectations and mental models that can make a theoretically sound app not usable by its intended users. There are to date few actively used dApps on any smart contract platform, and this might be because of a failure to communicate the platform's advantages to the broader audience. Our lottery scheme is an interactive one where users are required to be online and send transactions over a period of time, and possibly enact countermeasures to attacks by adversaries. This is much more complicated than traditional online gambling sites where one typically performs one transaction and is then automatically able to withdraw the potential earnings.

6 Conclusion

The work of this research project has been to implement a distributed lottery in Ethereum and analyze its properties and viability by gathering data through simulations and discussing adversarial threats. We found that a lottery similar to that of [22] can easily be implemented in smart contracts on Ethereum. Measurements on the transaction costs and demand for resources on the blockchain for setting up and playing the lottery were made, and this was used to estimate limits for the scalability and ticket prices of the implementation. Several common security concerns for blockchains were discussed in relation to the lottery, and we found the most concerning to probably be transaction censorship by a powerful miner or collusion of miners, particularly if they follow a selfish mining strategy.

Taking the current transaction throughput of the Ethereum blockchain into account, we found a limit to how many participants a lottery of this type is able to handle if it is to finish within a reasonable time period. This limit can be expressed as a linear function of the duration of the lottery, the total number of transactions in each block, and the ratio of total transactions on Ethereum that are addressed to the lottery. By setting the time limit to a few days and assuming that 10% of all transactions on Ethereum are addressed to the lottery, we found the limit to be about 100000 or 2^{17} participants.

The security of the lottery was discussed in the context of several known attack vectors for blockchain applications and web applications. We found that censorship of transactions from a powerful collusion of miners is likely to be the most concerning threat. Due to the interactivity and time limits of the lottery protocol, we found that if miners successfully block transactions from one account for each level of the tournament, they have the power to select an arbitrary willing participant to win the entire lottery. We discussed mitigating this issue, but were unable to find a satisfactory solution other than not playing the lottery with high stakes if there is an adversarial powerful miner collusion in the blockchain network.

Simulations of setting up and playing the lottery were executed on a local Ethereum blockchain to get insight into gas usage and whether our proof-of-concept implementation worked as expected. We found that the implementation does work as expected on a local blockchain, but attempts to deploy it on a global blockchain were not made. Gas usage was dominated by the smart contract deployment transactions needed to set up the lottery, which is all paid for by the lottery organizer. From this fact we assume that the lottery is viable only if the lottery organizer can

take a fee of the total prize that at least covers the expense from setting it up. Based on the transaction costs, we found a way to estimate a minimum ticket price as a function of gas price and organizer fee. We found that transaction costs can be reduced by 42% by splitting code into two separate match contracts, and that additional savings can probably be made by deduplicating code.

A shortened version of this work was submitted to the **2nd International Workshop on Future perspective of decentralized applications 2019** and is at the time of submitting this thesis awaiting review.

6.1 Future work

We have identified several directions for further research on implementing lotteries on smart contact platforms.

6.1.1 Minimizing transaction costs

We were able to reduce gas usage by 42% by splitting up the match contract to two separate contracts. As we mentioned earlier, it should be possible to reduce gas usage even further by deduplicating code by using patterns such as library contracts and proxy contracts. As of our implementation, the organizer has to take a risk when setting up the lottery, as there is a chance the lottery will fail to start by not enough participants joining. By reducing transaction costs, the potential losses from this risk will be lower, and the lottery might be played with lower ticket prices or lower house edge.

6.1.2 Minimizing interactivity

The lottery implemented in this thesis requires participants to interact by sending transactions several times. Other blockchain lottery schemes such as [20, 21] require less interaction, but do need a high deposit from all players. In our discussion on security we identified a vulnerability in attacks from miners censoring transactions. We also found that with the current transaction throughput in Ethereum, our lottery will need a very long time to finish if it scales to hundreds of thousands of participants. It might be possible to come up with a design that both require few interactions and does not need high deposits from participants to avoid halting.

6.1.3 Off-chain negotiation

Even though the smart contracts in our lottery must have the capability of enforcing the protocol, they don't necessarily need to process all the transactions as we laid out. In order to reduce gas usage and number of interactions, matches can be negotiated between opponents off-chain. For instance, when both players have made their commitments, they can reveal their secrets over another communication channel. By seeing one's opponent's secret, one knows what the outcome of

the match will be if both players were to reveal the secret on the blockchain. Instead, only the winner can reveal their secret while the loser forfeits by not making the reveal transaction. Even though the gas usage from playing the matches is relatively low compared to the overall gas usage, the number of transactions needed to play the matches is quite high. Decreasing the number of transactions needed to complete a match could improve the scalability as the blockchain has limited space for transactions per time unit.

It would be interesting to explore if hierarchical deterministic secrets could be used to avoid making a commitments for each match. HDS is a scheme where a private key can generate child private keys and a corresponding public key can generate child public keys. Since the algorithm used is deterministic, the generated secrets can possibly be verified at a later stage. While the original HDS scheme proposed in [66] might be unfit for the purpose, as leaked child keys can make it possible to derive the parent private key, the scheme proposed in [68] does not have this vulnerability and might be better suited.

6.1.4 Formal analysis of security

While we have discussed security issues in this thesis, we believe a more rigorous analysis is needed. As the blockchain space matures, more research and experience is likely to appear. This can be used to arrive at better estimates on what assumptions and conditions are needed to make our lottery secure. Perhaps most important is more knowledge on the topic of censorship of transactions. Both models that allow us to find under what conditions censorship is likely, as well as tactics to avoid censorship would be interesting to explore in the context of a distributed lottery on a blockchain.

Bibliography

[1] Sintomer, Y. 2010. Random Selection, Republican Self-Government, and Deliberative Democracy. *Constellations*, 17(3), 472–487. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8675.2010.00607.x, doi:10.1111/j.1467-8675.2010.00607.x.

[2] Nixon, R. M. November 1969. Executive Order 11497—Amending the Selective Service Regulations to Prescribe Random Selection. URL: https://web.archive.org/web/20180922121238/http://www.presidency.ucsb.edu/ws/?pid=106002.

[3] Economist, T. April 2018. Why a licence plate costs more than a car in Shanghai. *The Economist*. URL: https://www.economist.com/china/2018/04/19/why-a-licence-plate-costs-more-than-a-car-in-shanghai.

[4] Sako, K. 1999. Implementation of a Digital Lottery Server on WWW. In *Secure Networking — CQRE [Secure] ' 99*, Baumgart, R., ed, Lecture Notes in Computer Science, 101–108. Springer Berlin Heidelberg.

[5] Konstantinou, E., Liagkou, V., Spirakis, P., Stamatiou, Y. C., & Yung, M. 2004. Electronic National Lotteries. In *Financial Cryptography*, Juels, A., ed, Lecture Notes in Computer Science, 147–163. Springer Berlin Heidelberg.

[6] Konstantinou, E., Liagkou, V., Spirakis, P., Stamatiou, Y., & Yung, M. 2005. "Trust engineering:" from requirements to system design and maintenance - A working national lottery system experience. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3650 LNCS, 44–58.

[7] Chen, Y.-Y., Jan, J.-K., & Chen, C.-L. 2005. Design of a fair proxy raffle protocol on the Internet. *Computer Standards and Interfaces*, 27(4), 415–422. doi:10.1016/j.csi.2004.11.002.

[8] Kuacharoen, P. 2012. Design and Implementation of a Secure Online Lottery System. In *Advances in Information Technology*, Papasratorn, B., Charoenkitkarn, N., Lavangnananda, K., Chutimaskul, W., & Vanijja, V., eds, Communications in Computer and Information Science, 94–105. Springer Berlin Heidelberg.

[9] Chen, C.-L., Chiang, M.-L., Lin, W.-C., & Li, D.-K. 2016. A novel lottery protocol for mobile environments. *Computers and Electrical Engineering*, 49, 146–160. doi:10.1016/j.compeleceng.2015.07.016.

[10] Douceur, J. R. 2002. The sybil attack. In *International workshop on peer-to-peer systems*, 251–260. Springer.

[11] Shamir, A., Rivest, R. L., & Adleman, L. M. 1981. Mental poker. In *The mathematical gardner*, 37–43. Springer.

[12] Blum, M. 1983. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1), 23–27.

[13] Broder, A. 1985. A provably secure polynomial approximation scheme for the distributed lottery problem (Extended abstract). 136–148. doi:10.1145/323596.323608.

[14] Goldreich, O., Micali, S., & Wigderson, A. January 1987. How to play ANY mental game. 218–229. doi:10.1145/28395.28420.

[15] Goldschlag, D. & Stubblebine, S. 1998. Publicly verifiable lotteries: Applications of delaying functions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1465, 214–226. doi:10.1007/BFb0055485.

[16] Fouque, P.-A., Poupard, G., & Stern, J. 2001. Sharing decryption in the context of voting or lotteries. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1962, 90–104.

[17] Crosby, M., Pattanayak, P., Verma, S., & Kalyanaraman, V. 2016. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10), 71. URL: https://j2-capital.com/wp-content/uploads/2017/11/AIR-2016-Blockchain.pdf.

[18] Back, A. & Bentov, I. February 2014. Note on fair coin toss via Bitcoin. URL: https://arxiv.org/abs/1402.3698v1.

[19] Andrychowicz, M., Dziembowski, S., Malinowski, D., & Mazurek, L. 2014. Fair two-party computations via bitcoin deposits. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8438, 105–121. doi:10.1007/978-3-662-44774-1_8.

[20] Andrychowicz, M., Dziembowski, S., Malinowski, D., & Mazurek, L. 2014. Secure multiparty computations on bitcoin. 443–458. doi:10.1109/SP.2014.35.

[21] Bentov, I. & Kumaresan, R. 2014. How to use Bitcoin to design fair protocols. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8617 LNCS(PART 2), 421–439. doi:10.1007/978-3-662-44381-1_24.

[22] Miller, A. & Bentov, I. 2017. Zero-collateral lotteries in Bitcoin and Ethereum. 4–13. doi:10.1109/EuroSPW.2017.44.

[23] Bartoletti, M. & Zunino, R. 2017. Constant-deposit multiparty lotteries on bitcoin. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10323 LNCS, 231–247. doi:10.1007/978-3-319-70278-0_15.

[24] Lindell, Y. & Katz, J. 2014. *Introduction to modern cryptography*. Chapman and Hall/CRC.

[25] Wang, X., Lai, X., Feng, D., Chen, H., & Yu, X. 2005. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *Advances in Cryptology – EUROCRYPT 2005*, Cramer, R., ed, Lecture Notes in Computer Science, 1–18. Springer Berlin Heidelberg.

[26] Merkle, R. C. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Pomerance, C., ed, Lecture Notes in Computer Science, 369–378. Springer Berlin Heidelberg.

[27] Brassard, G., Chaum, D., & Crépeau, C. 1988. Minimum disclosure proofs of knowledge. *Journal of computer and system sciences*, 37(2), 156–189.

[28] Dwork, C. & Naor, M. 1993. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology — CRYPTO' 92*, Brickell, E. F., ed, Lecture Notes in Computer Science, 139–147. Springer Berlin Heidelberg.

[29] Back, A. 2002. Hashcash-a denial of service counter-measure.

[30] Diffie, W. & Hellman, M. 1976. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6), 644–654.

[31] Micali, S., Rabin, M., & Vadhan, S. October 1999. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 120–130. doi:10.1109/SFFCS.1999.814584.

[32] Boneh, D., Bonneau, J., Bünz, B., & Fisch, B. 2018. Verifiable Delay Functions. In *Advances in Cryptology – CRYPTO 2018*, Shacham, H. & Boldyreva, A., eds, Lecture Notes in Computer Science, 757–788. Springer International Publishing.

[33] Syverson, P. 1998. Weakly secret bit commitment: applications to lotteries and fair exchange. 2–13.

[34] Rivest, R. L., Shamir, A., & Wagner, D. A. Time-lock Puzzles and Timed-release Crypto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[35] Rabin, M. O. October 1983. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2), 256–267. URL: http://www.sciencedirect.com/science/article/pii/0022000083900429, doi:10.1016/0022-0000(83)90042-9.

[36] Clark, J. & Hengartner, U. 2010. On the Use of Financial Data as a Random Beacon. *EVT/WOTE*, 89.

[37] Bentov, I., Gabizon, A., & Zuckerman, D. May 2016. Bitcoin Beacon. URL: https://arxiv.org/abs/1605.04559v2.

[38] Yajam, H., Ebadi, E., Badakhshan, M., & Akhaee, M. 2019. Improvement on Bitcoin's Verifiable Public Randomness with Semi-Trusted Delegates. 53–57. doi:10.1109/ISTEL.2018.8661008.

[39] Bonneau, J., Clark, J., & Goldfeder, S. 2015. On Bitcoin as a public randomness source. *IACR Cryptology ePrint Archive*, 2015, 1015.

[40] Pierrot, C. & Wesolowski, B. 2018. Malleability of the blockchain's entropy. *Cryptography and Communications*, 10(1), 211–233. doi:10.1007/s12095-017-0264-3.

[41] NIST. May 2019. NIST Randomness Beacon | NIST. URL: https://web.archive.org/web/20190509161828/https://www.nist.gov/programs-projects/nist-randomness-beacon.

[42] Randao. URL: https://github.com/randao/randao/commits/master.

[43] Chatterjee, K., Goharshady, A. K., & Pourdamghani, A. February 2019. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. *arXiv:1902.07986 [cs]*. arXiv: 1902.07986. URL: http://arxiv.org/abs/1902.07986.

[44] Zhou, J. & Tan, C. 2001. Playing lottery on the internet. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2229, 189–201.

[45] Chow, S., Hui, L., Yiu, S., & Chow, K. 2005. An e-lottery scheme using Verifiable Random Function. volume 3482, 651–660.

[46] Lee, J.-S., Chang, C.-C., & Fellow, I. February 2009. Design of electronic t-out-of-n lotteries on the Internet. *Computer Standards & Interfaces*, 31(2), 395–400. URL: http://www.sciencedirect.com/science/article/pii/S0920548908000731, doi:10.1016/j.csi.2008.05.004.

[47] Liu, Y., Lin, D., Cheng, C., Chen, H., & Jiang, T. 2014. An improved t-out-of-n e-lottery protocol. *International Journal of Communication Systems*, 27(11), 3223–3231. doi:10.1002/dac.2536.

[48] Xia, Z., Liu, Y., Hsu, C.-F., & Chang, C.-C. 2019. An information theoretically secure e-lottery scheme based on symmetric bivariate polynomials. *Symmetry*, 11(1). doi:10.3390/sym11010088.

[49] Grumbach, S. & Riemann, R. 2017. Distributed random process for a large-scale peer-to-peer lottery. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10320 LNCS, 34–48. doi:10.1007/978-3-319-59665-5_3.

[50] Goldschlag, D., Stubblebine, S., & Syverson, P. 2010. Temporarily hidden bit commitment and lottery applications. *International Journal of Information Security*, 9(1), 33–50. doi:10.1007/s10207-009-0094-1.

[51] SmartBillions. SmartBillions ICO. https://www.smartbillions.com/. URL: https://www.smartbillions.com/.

[52] Fairlotto. July 2018. Fairlotto V2: Provably Fair STEEM Blockchain Based Lottery - All New Front-End + Features. https://steemit.com/lottery/@fairlotto/fairlotto-v2-provably-fair-steem-blockchain-based-lottery-all-new-front-e URL: https://steemit.com/lottery/@fairlotto/fairlotto-v2-provably-fair-steem-blockchain-based-lottery-all-new-front-e

[53] SatoshiDice. Satoshi Dice Bitcoin Games | Bitcoin Dice Game | SatoshiDICE. https://web.archive.org/web20190401184324/http://satoshidice.com/. URL: https://web.archive.org/web20190401184324/http://satoshidice.com/.

[54] Tschorsch, F. & Scheuermann, B. 2016. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys Tutorials*, 18(3), 2084–2123. doi:10.1109/COMST.2016.2535718.

[55] Garay, J., Kiayias, A., & Leonardos, N. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology - EUROCRYPT 2015*, Oswald, E. & Fischlin, M., eds, Lecture Notes in Computer Science, 281–310. Springer Berlin Heidelberg.

[56] Nakamoto, S. 2008. Bitcoin: A peer-to-peer electronic cash system. doi:https://bitcoin.org/bitcoin.pdf.

[57] Andresen, G. March 2013. March 2013 Chain Fork Post-Mortem. original-date: 2013-11-19T17:18:41Z. URL: https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki.

[58] Buterin, V. June 2015. The Problem of Censorship. URL: https://blog.ethereum.org/2015/06/06/the-problem-of-censorship/.

[59] Wuille, P. May 2019. security - 51% attack - apparently very easy? refering to CZ's "rollback btc chain" - How to make sure such corruptible scenario can never happen so easily? URL: https://bitcoin.stackexchange.com/questions/87652/51-attack-apparently-very-easy-refering-to-czs-rollback-btc-chain-how-t/87655.

[60] Eyal, I. & Sirer, E. G. June 2018. Majority is Not Enough: Bitcoin Mining is Vulnerable. *Commun. ACM*, 61(7), 95–102. URL: http://doi.acm.org/10.1145/3212998, doi:10.1145/3212998.

[61] Szabo, N. September 1997. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9). URL: https://firstmonday.org/ojs/index.php/fm/article/view/548, doi:10.5210/fm.v2i9.548.

[62] Wood, G. 2018. Ethereum: a secure decentralised generalised transaction ledger. byzantium version (2018-06-05).

[63] Lu, A. May 2018. Solidity DelegateProxy Contracts. URL: https://blog.gnosis.pm/solidity-delegateproxy-contracts-e09957d0f201.

[64] Walsh, J. The House's Edge In Lotto | Easy Money | FRONTLINE | PBS. URL: https://www.pbs.org/wgbh/pages/frontline/shows/gamble/odds/house.html.

[65] Shackleford, M. House Edge of casino games compared. URL: https://wizardofodds.com/gambling/house-edge/.

[66] Wuille, P. February 2012. Bitcoin Improvement Proposals. Contribute to bitcoin/bips development by creating an account on GitHub. original-date: 2013-11-19T17:18:41Z. URL: https://github.com/bitcoin/bips.

[67] Dhillon, V., Metcalf, D., & Hooper, M. 2017. The DAO Hacked. In *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You*, Dhillon, V., Metcalf, D., & Hooper, M., eds, 67–78. Apress, Berkeley, CA. URL: https://doi.org/10.1007/978-1-4842-3081-7_6, doi:10.1007/978-1-4842-3081-7_6.

[68] Gutoski, G. & Stebila, D. 2015. Hierarchical Deterministic Bitcoin Wallets that Tolerate Key Leakage. In *Financial Cryptography and Data Security*, Böhme, R. & Okamoto, T., eds, Lecture Notes in Computer Science, 497–504. Springer Berlin Heidelberg.

A Listings

A.1 Solidity contracts

A.1.1 Abstract match contract

```
pragma solidity >=0.5.0 <0.6.0;

/**
 * Match as a digital coin toss between two players.
 * A tournament is made from a tree of these matches.
 **/
contract AbstractLotteryMatch {

    /**
     * Have a player commit to a value for the digital coin toss.
     */
    function commit(bytes32 _c) public;

    /**
     * Have a player reveal the value previously commited to for the digital coin toss.
     */
    function reveal(uint256 _s) public;

    /**
     * Implicitly calculate the winner by performing the digital coin toss.
     */
    function getWinner() public view returns (address winner);
}
```

Listing A.1: Full Solidity contract for AbstractLotteryMatch.

A.1.2 First level match contract

```
pragma solidity >=0.5.0 <0.6.0;

import { LotteryMaster } from "./LotteryMaster.sol";
import { AbstractLotteryMatch } from "./AbstractLotteryMatch.sol";

/**
 * Match as a digital coin toss between two players.
 * A tournament is made from a tree of these matches.
 **/
contract FirstLevelMatch is AbstractLotteryMatch {

    address public alice;  // Player 1 of the match.
    address public bob;  // Player 2 of the match.

    mapping(address => bytes32) public commitments;  // Commitments alice and bob have made.
```

```
16      mapping(address => uint256) public secrets;  // Secrets, which are
            preimages to the commitments, alice and bob have made.
17
18      LotteryMaster public lottery;  // The master lottery contract.
19      uint256 public index;  // Matches on the first level are indexed
            so that they map to specific players in the lottery.
20
21      uint256 public tCommit;  // Block height after which making
            commitments is possible.
22      uint256 public tReveal;  // Block height after which making
            reveals is possible. And commitments no longer possible.
23      uint256 public tPlay;  // Block height after which deciding the
            winner is possible. And reveals no longer possible.
24
25      constructor(uint256 _tCommit, uint256 _tReveal, uint256 _tPlay,
            LotteryMaster _lottery, uint256 _index) public {
26          require(_tCommit < _tReveal, "Invalid time limits. tCommit not
                before tReveal.");
27          require(_tReveal < _tPlay, "Invalid time limits. tReveal not
                before tPlay.");
28
29          tCommit = _tCommit;
30          tReveal = _tReveal;
31          tPlay = _tPlay;
32
33          lottery = _lottery;
34          index = _index;
35      }
36
37      /**
38       * Have a player commit to a value for the ditial coin toss.
39       */
40      function commit(bytes32 _c) public {
41          require(tCommit < block.number, "Too early to commit.");
42          require(tReveal > block.number, "Too late to commit.");
43
44          alice = lottery.getPlayer(index * 2);
45          bob = lottery.getPlayer(index * 2 + 1);
46          require(msg.sender == alice || msg.sender == bob, "Wrong
                player for this match.");
47          require(commitments[msg.sender] == 0, "Player has already
                commited to this match.");
48
49          commitments[msg.sender] = _c;
50      }
51
52      /**
53       * Have a player reveal the value previously commited to for the
            digital coin toss.
54       */
55      function reveal(uint256 _s) public {
56          require(tReveal < block.number, "Too early to reveal.");
57          require(tPlay > block.number, "Too late to reveal.");
58
59          require(keccak256(abi.encodePacked(msg.sender, _s)) ==
                commitments[msg.sender], "Secret not preimage of
                commitment.");
```

```
60
61          secrets[msg.sender] = _s;
62      }
63
64      /**
65       * Implicitly calculate the winner by performing the digital coin
             toss.
66       */
67      function getWinner() public view returns (address winner) {
68          require(tPlay < block.number, "Too early to determine a winner
                .");
69
70          // Check if any player is missing
71          if (alice != address(0) && bob == address(0)) {
72              return alice;
73          } else if (alice == address(0) && bob != address(0)) {
74              return bob;
75          } else if (alice == address(0) && bob == address(0)) {
76              return lottery.getPlayer(index * 2);
77          }
78
79          // Check if parties have made commitments.
80          if (commitments[alice] != 0 && commitments[bob] == 0) {
81              return alice;
82          } else if (commitments[alice] == 0 && commitments[bob] != 0) {
83              return bob;
84          } else if (commitments[alice] == 0 && commitments[bob] == 0) {
85              return lottery.getPlayer(index * 2);
86          }
87
88          // Check if parties have revealed.
89          if (secrets[alice] != 0 && secrets[bob] == 0) {
90              return alice;
91          } else if (secrets[alice] == 0 && secrets[bob] != 0) {
92              return bob;
93          } else if (secrets[alice] == 0 && secrets[bob] == 0) {
94              return lottery.getPlayer(index * 2);
95          }
96
97          // Both parties have revealed, let's toss the coin.
98          if ((secrets[alice] ^ secrets[bob]) % 2 == 0) {
99              return alice;
100         } else {
101             return bob;
102         }
103     }
104 }
```

Listing A.2: Full Solidity contract for FirstLevelMatch.

A.1.3 Internal match contract

```
1  pragma solidity >=0.5.0 <0.6.0;
2
3  import { LotteryMaster } from "./LotteryMaster.sol";
4  import { AbstractLotteryMatch } from "./AbstractLotteryMatch.sol";
5
6  /**
```

```
7   * Match as a digital coin toss between two players.
8   * A tournament is made from a tree of these matches.
9   **/
10  contract InternalMatch is AbstractLotteryMatch{
11
12      address public alice;  // Player 1 of the match.
13      address public bob;  // Player 2 of the match.
14
15      mapping(address => bytes32) public commitments;  // Commitments
            alice and bob have made.
16      mapping(address => uint256) public secrets;  // Secrets, which are
             preimages to the commitments, alice and bob have made.
17
18      AbstractLotteryMatch public left;  // One of the matches for
            qualifying to this match. A contract address.
19      AbstractLotteryMatch public right;  // One of the matches for
            qualifying to this match. A contract address.
20
21      uint256 public tCommit;  // Block height after which making
            commitments is possible.
22      uint256 public tReveal;  // Block height after which making
            reveals is possible. And commitments no longer possible.
23      uint256 public tPlay;  // Block height after which deciding the
            winner is possible. And reveals no longer possible.
24
25      constructor(uint256 _tCommit, uint256 _tReveal, uint256 _tPlay,
            AbstractLotteryMatch _left, AbstractLotteryMatch _right)
            public {
26          require(_tCommit < _tReveal, "Invalid time limits. tCommit not
                 before tReveal.");
27          require(_tReveal < _tPlay, "Invalid time limits. tReveal not
                before tPlay.");
28
29          tCommit = _tCommit;
30          tReveal = _tReveal;
31          tPlay = _tPlay;
32
33          left = _left;
34          right = _right;
35      }
36
37
38      /**
39       * Have a player commit to a value for the ditial coin toss.
40       */
41      function commit(bytes32 _c) public {
42          require(tCommit < block.number, "Too early to commit.");
43          require(tReveal > block.number, "Too late to commit.");
44
45          alice = left.getWinner();
46          bob = right.getWinner();
47          require(msg.sender == alice || msg.sender == bob, "Wrong
                player for this match.");
48          require(commitments[msg.sender] == 0, "Player has already
                commited to this match.");
49
50          commitments[msg.sender] = _c;
```

```
51      }
52
53      /**
54       * Have a player reveal the value previously commited to for the
             digital coin toss.
55       */
56      function reveal(uint256 _s) public {
57          require(tReveal < block.number, "Too early to reveal.");
58          require(tPlay > block.number, "Too late to reveal.");
59
60          require(keccak256(abi.encodePacked(msg.sender, _s)) ==
                commitments[msg.sender], "Secret not preimage of
                commitment.");
61
62          secrets[msg.sender] = _s;
63      }
64
65      /**
66       * Implicitly calculate the winner by performing the digital coin
             toss.
67       */
68      function getWinner() public view returns (address winner) {
69          require(tPlay < block.number, "Too early to determine a winner
                .");
70
71          // Check if any player is missing
72          if (alice != address(0) && bob == address(0)) {
73              return alice;
74          } else if (alice == address(0) && bob != address(0)) {
75              return bob;
76          } else if (alice == address(0) && bob == address(0)) {
77              return left.getWinner();
78          }
79
80          // Check if parties have made commitments.
81          if (commitments[alice] != 0 && commitments[bob] == 0) {
82              return alice;
83          } else if (commitments[alice] == 0 && commitments[bob] != 0) {
84              return bob;
85          } else if (commitments[alice] == 0 && commitments[bob] == 0) {
86              return left.getWinner();
87          }
88
89          // Check if parties have revealed.
90          if (secrets[alice] != 0 && secrets[bob] == 0) {
91              return alice;
92          } else if (secrets[alice] == 0 && secrets[bob] != 0) {
93              return bob;
94          } else if (secrets[alice] == 0 && secrets[bob] == 0) {
95              return left.getWinner();
96          }
97
98          // Both parties have revealed, let's toss the coin.
99          if ((secrets[alice] ^ secrets[bob]) % 2 == 0) {
100             return alice;
101         } else {
102             return bob;
```

```
        }
    }
}
```

Listing A.3: Full Solidity contract for InternalMatch.

A.1.4 Master contract

```
pragma solidity >=0.5.0 <0.6.0;

import { AbstractLotteryMatch } from "./AbstractLotteryMatch.sol";

/**
 * The lottery master contract which individual 1v1 matches reference.
 */
contract LotteryMaster {

    address[] public players;  // All players who have made a deposit.
    mapping(address => uint256) public deposits;  // The value of
        deposits players have made.

    address public owner;  // Owner of this contract.

    uint256 public price;  // Price in wei for buying a ticket.
    uint256 public N;  // Max number of players in the lottery.

    uint256 public nPlayers;  // Number of players currently joined.

    uint256 public tStart;  // Start block height of the lottery.

    AbstractLotteryMatch public finalMatch;  // Reference to the final
         match which decides the winner.

    bool public isInitialized;  // Whether the lottery is ready to
        take deposits.
    bool public isFull;  //  Whether the lottery is full and ready to
        play.

    constructor(uint256 _N, uint256 _price, uint256 _tStart) public {
        require(_tStart < block.number, "Time limits invalid start
            time is in the past.");

        N = _N;
        price = _price;
        tStart = _tStart;

        owner = msg.sender;
    }

    /**
     * Set the final match of the lottery.
     * The lottery should not be able to start before this is set.
     * It's up to participants to validate that this final match is
         the correct contract.
     */
    function setFinalMatch(AbstractLotteryMatch _finalMatch) public {
        require(msg.sender == owner, "Only owner can set final match."
            );
```

```
44          require(finalMatch == AbstractLotteryMatch(0), "Final match is
                already set.");
45          finalMatch = _finalMatch;
46
47          isInitialized = true;
48      }
49
50
51      /**
52       * Players can make a deposit to join the lottery. This is
53       * equivalent to buying a ticket.
54       */
55      function deposit() public payable {
56          require(block.number < tStart, "Too late to deposit now.");
57          require(isInitialized == true, "Final match not set. Lottery
                not initialized yet.");
58          require(msg.value == price, "Transaction value is not equal to
                ticket price.");
59          require(isFull == false, "Lottery is full");
60          require(deposits[msg.sender] == 0, "Player has already
                deposited to this lottery.");
61
62          players.push(msg.sender);
63          deposits[msg.sender] = msg.value;
64          nPlayers++;
65
66          if (nPlayers == N) {
67              isFull = true;
68          }
69      }
70
71      /**
72       * After the predetermined end time of the lottery has passed,
             then either
73       * (a) the winner can withdraw their prize, or (b) there is no
             winner and
74       * participants can withdraw their deposit.
75       */
76      function withdraw() public {
77
78          if (block.number >= tStart && !isFull) {
79              // Lottery did not get enough participants, so
                    participants can withdraw their deposit.
80              msg.sender.transfer(deposits[msg.sender]);
81          } else {
82              // The winner can withdraw their prize.
83              address lotteryWinner = finalMatch.getWinner();
84              require(msg.sender == lotteryWinner, "Player is not winner
                     of lottery.");
85              msg.sender.transfer(address(this).balance);
86          }
87      }
88
89      function getPlayer(uint256 index) public view returns (address
            player) {
90          player = players[index];
91      }
```

```
92
93      /**
94       * The only way to get an array is to make a function to get it.
95       */
96      function getPlayers() public view returns (address[] memory
            _players) {
97          _players = players;
98      }
99  }
```

Listing A.4: Full Solidity contract for LotteryMaster.

A.2 Javascript

A.2.1 Simulate lottery setup

```
1  contract('Simulate lottery build', (accounts) => {
2    it('Should build master contract and match contracts for n players',
         async () => {
3      const L = 2;
4      const N = 2 ** L;
5
6      const organizerAddress = accounts[0];
7      const organizerInitialBalance = web3.utils.fromWei(
8        await web3.eth.getBalance(organizerAddress),
9        'ether'
10     );
11
12     console.log(`Organizer has ${organizerInitialBalance} ether`);
13
14     console.log(`Building lottery with ${N} players.`);
15     const startTime = new Date();
16     const lotteryBuilder = new LotteryBuilder(N, price, tStart, tFinal
           , td);
17     await lotteryBuilder.start();
18     console.log(`Built lottery in ${new Date() - startTime} ms`);
19
20     const organizerFinalBalance = web3.utils.fromWei(
21       await web3.eth.getBalance(organizerAddress),
22       'ether'
23     );
24
25     console.log(`Organizer has ${organizerFinalBalance} ether`);
26     console.log(
27       `Organizer used ${organizerInitialBalance -
28         organizerFinalBalance} ether for gas.`
29     );
30   });
31 });
```

Listing A.5: Truffle test suite for simulating lottery setup.

A.2.2 Simulate lottery play

```
1  contract('Simulate lottery play', (accounts) => {
2    it('Should play correctly', async () => {
3      const L = 2;
4      const N = 2 ** L;
```

```
5
6       console.log(`Simulating lottery with ${N} players.`);
7       let startTime = new Date();
8       let tempTime = new Date();
9
10      const lotteryBuilder = new LotteryBuilder(N, price, tStart, tFinal
            , td);
11      await lotteryBuilder.start();
12
13      console.log(`Built lottery in ${new Date() - tempTime} ms`);
14      tempTime = new Date();
15
16      const lottery = new LotteryContract(lotteryBuilder.lottery.address
            );
17      await lottery.init();
18
19      console.log(
20        `Initialized lottery playing contract in ${new Date() - tempTime
              } ms`
21      );
22      tempTime = new Date();
23
24      const matches = await lottery.getAllMatches();
25
26      console.log(`Got all lottery matches in ${new Date() - tempTime}
            ms`);
27      tempTime = new Date();
28
29      const players = generatePlayers(N, accounts);
30      const playerMap = players.reduce(
31        (acc, x) => ({ ...acc, [x.address]: x }),
32        {}
33      );
34
35      for (const { address } of players) {
36        await lottery.deposit(address);
37      }
38
39      console.log(`All players joined in ${new Date() - tempTime} ms`);
40      tempTime = new Date();
41
42      const contractPlayers = await lottery.getPlayers();
43
44      let winners = contractPlayers.map((address) => playerMap[address])
            ;
45      let level = 0;
46
47      while (winners.length > 1) {
48        console.log(`Playing level ${level}.`);
49        for (const [i, { address, commitment }] of winners.entries()) {
50          await matches[level][i >> 1].commit(commitment, { from:
                address });
51        }
52        for (const [i, { address, secret }] of winners.entries()) {
53          await matches[level][i >> 1].reveal(secret, { from: address })
                ;
54        }
```

```
55        winners = await Promise.all([
56          ...matches[level].map(async (match) => {
57            return playerMap[await match.getWinner()];
58          }),
59        ]);
60        level++;
61        console.log(`Played level ${level} in ${new Date() - tempTime}
              ms`);
62        tempTime = new Date();
63      }
64      console.log(`Played lottery in ${new Date() - startTime} ms`);
65
66      const winner = await lottery.getWinner();
67      assert.notEqual(winner, ZERO_ADDRESS);
68      assert.equal(winner, winners[0].address);
69
70      await lottery.lotteryContract.withdraw({ from: winner });
71
72      console.log(
73        `Winner is ${winner} who now has ${web3.utils.fromWei(
74          await web3.eth.getBalance(winner),
75          'ether'
76        )} eth`
77      );
78    });
79  });
```

Listing A.6: Truffle test suite for simulating lottery play.

B Simulation data

B.1 Gas usage

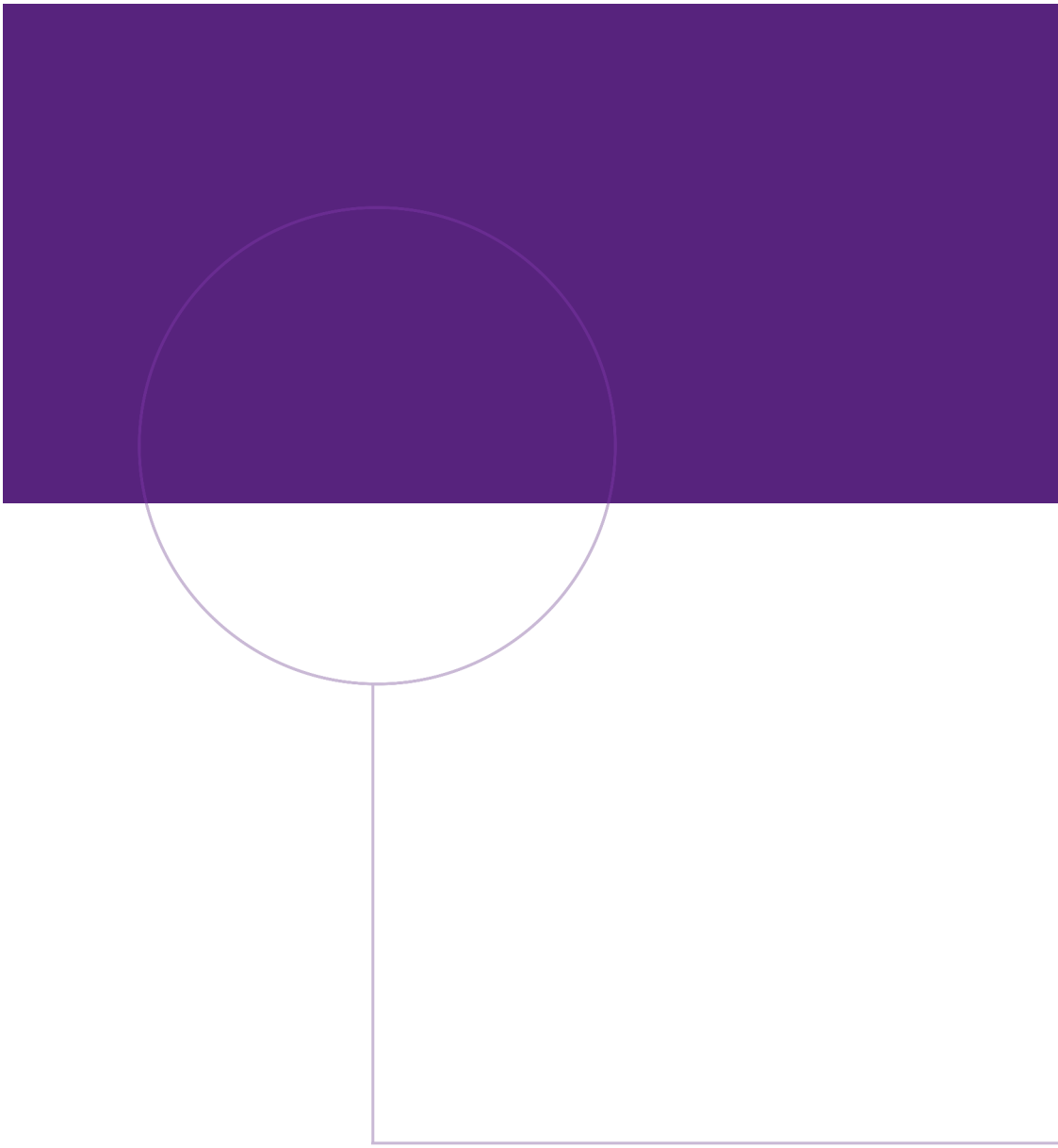
Steps to replicate

Using Git repository located at https://github.com/viktorfa/lottery-truffle.git. Dual match simulations from Git commit `36ee7818`. Single match simulations from Git commit `63747577`. Variable `L` in `tests/Lottery.simulate.js` set to 8 for both simulation suites. Simulations run with command `truffle test ./test/Lottery.simulate.js` with a Ganache Ethereum RPC client running.

Results

Table 11: Statistics from dual match simulation with 256 participants.

Stats n=5					
Contract	Method	Mean	Stdev	Range	Stdev/Mean
FirstLevelMatch	commit	76611	0	0	0
FirstLevelMatch	reveal	42883	0.8367	1.25	0.00001951
InternalMatch	commit	88672	2.51	7	0.00002831
InternalMatch	reveal	42883	1.673	4	0.00003902
LotteryMaster	deposit	73925	0	0	0
LotteryMaster	setFinalMatch	49282	0	0	0
LotteryMaster	withdraw	39171	5.477	10	0.0001398
Deployments					
FirstLevelMatch		1693715	28.62	64	0.0000169
InternalMatch		1697002	1830	1026	0.0010785
LotteryMaster		1693664	0	0	0
Total set up		433555262	3534	2288	0.00000815
Total set up and play		516524423	3428	2917	0.00000664



NTNU
Norwegian University of
Science and Technology