TrapLottery 0.2: Automated Lottery on the Blockchain

Description "Trustless Lottery w/ Verifiable Delay Functions"
Author Mario A. Barbara [1]
Working Implementation https://kovan.etherscan.io/address/0x23cacafb1f32c62b805b14fac3417f1e1e3fec98
Disclaimer *Please check out the PDF format of this README.*

In this project I built an automated lottery on the Blockchain, ensuring soundness through the exclusive use of cryptography. The main cryptographic primitive is Verifiable Delay Functions (VDF), and the project takes form of a "server" contract on the Ethereum "Kovan" test network, written in Solidity. The client is, instead, implemented in Python (since some computations would be too heavy for the contract itself).

1) The Crypto: Verifiable Delay Functions

Main Concept

The cryptographic concept of "Time-Lock Puzzles" was originally introduced by [RSW96], and previously touched upon using more naive techniques [Merkle78]. The core concept was to have someone compute huge exponentiations under a group of unknown order. For the RSA group, this implies not being able to reduce a huge exponent (power of 2) to lie within the Totient's bounds, thus forcing repeated squarings within the group until the result is achieved:

$$e = 2^T \pmod{\phi(N)}, \quad y = x^e \pmod{N} \ \textit{(order is known)}$$
$$x \to x^2 \to x^{2^2} \to x^{2^3} \to ... \to x^{2^T} \pmod{N} \ \textit{(order is unknown)}$$

This idea was envisioned to have many applications:

- key-escrow for self-decrypting messages $\implies$ scheduling of future transactions
- self-revealing commitments $\implies$ self-revealing voting or bidding
- slow and sequential hash functions $\implies$ unbiased Randomness Beacon on the Blockchain
- and so on...

Properties

"Time-Lock Puzzles" were later extended by [BBBF18], introducing the concept of succint verifiability (or succint "Proof of Work") of the results, under the name "Verifiable Delay Functions". VDFs typically aim to have the following properties:

1) **Completeness** all valid outputs y and proofs π are always accepted. Can be enforced by contract logic.
2) **Soundness** all invalid outputs y and proofs π are always rejected. Can be enforced through cryptography, its security relies on the group's security assumptions.
3) **Sequentiality** any good VDF must guarantee that parallelism has near to no effect on the time spent computing the result. Repeated squaring under groups of unknown order is not believed to be parallelizable.
4) **Transparency** the setup of a secure VDF must guarantee a group of unknown order. Currently, there are multiple ideas on how to do this, but standard RSA group generation does **not** suffice.
5) **Proof Uniqueness ("Blockchainability")** allowing the VDF to accept multiple valid proofs is believed to lead to potential "grinding attacks". We can use the FiatShamir heuristic to always accept a single unique proof.
6) **Proof Succintness** it must be possible to validate π much faster than a normal evaluation of y.

As mentioned, the group needs to be of unknown order. In my implementation, I will be using an RSA group of order that is unknown to all but the creator of the group. Hence, it is more accurate to call it a "(Trapdoor) Verifiable Delay Function", because the owner of the RSA private key can always compute outputs really quickly. To make the Trapdoor VDF more transparent, some strategies are being currently discussed, and none has won over the other:

i) **Random N** This [Sander99] approach is the easiest and most efficient to apply, and argues that it can generate a modulus which very likely satisfies RSA requirements to some extent. However, it also severely damages VDF sequentiality requirements and can lead to more efficient exponentiation implementations.

[1] 16 February 2019 @ Eötvös Loránd University (ELTE), Budapest

ii) **MPC** Multi Party Computation typically employs a lot of resources and allows multiple parties to jointly generate a provably valid RSA modulus. This was also the approach taken by the Zcash blockchain. Unfortunately, the security of this approach lies on the existence of at least one honest party involved in the initial setup, which does not imply real transparency for all parties joining the blockchain later on.

iii) **Class Group of an Imaginary Quadratic Field** groups generated using this [BBHM02] approach promises to be the only non-compromising one. While this field of research is not new, it has not yet been studied enough to fully trust it.

Proof Schemes

We will discuss two proof generation schemes that yield working Trapdoor VDF implementations: [Wes18] and [Piet18]. They have also been discussed in [BBF18], which was the main source for this project.

1) **Pietrzak's proof**

the approach is to prove the condition $y = x^{2^T} \pmod N$ by using an Interactive Proof to show that the prover has also necessarily calculated the sq.root (in $\mod \phi(N)$) of y divided by 2. After this proof, the prover and verifier take the previous values (x, y) and change them to recursively engage in another Interactive Proof to prove that the previous root is also known, and leads to the current root of y. This process is repeated, and after $log_2(T)$ rounds, the final round can be completely verified by both parties in constant time, and is directly linked to x.

The proof can be considered to be $\pi = (\pi_1, \pi_2, ..., \pi_{log(T)})$, each round validates π_i:

$$P \to V : \pi_i \leftarrow x_i^{2^{(\frac{T_i}{2}-1)}} \pmod N$$
$$P \leftarrow V : r_i \xleftarrow{\$} \mathbb{Z}_\lambda \text{ } (\lambda \textit{ is the security parameter for the whole system. note that } \lambda \neq \lambda_{RSA})$$
$$P, V : y_{i+1} \leftarrow \pi_i^{r_i} y_i, \quad x_{i+1} \leftarrow (x_i^{r_i} \pi_i)^{2^{\frac{T_i}{2}}}, \quad x_{i+1} \stackrel{?}{=} y_{i+1}, \quad T_{i+1} \leftarrow T/2$$

2) **Wesolowski's proof**

this approach is closer to classic interactive proof systems, where the prover finds an alternative problem isomorphic to the previous one. In our scenario, showing that we have calculated $x^{2^T} \pmod N$ is also equivalent to showing that we have calculated *any* step in-between, thus reassuring the verifier that we must know the final result. Wesolowski, then, chooses a random challenge r used to decompose the exponent $2^T = r \cdot k + residue$: the remaining proof consists in showing that the r-th root (in $\mod \phi(N)$) of y is tied to x and y through these values. The challenge is also chosen to be prime, for security requirements.

The proof π can be found in the protocol as follows:

$$P \to V : y$$
$$P \leftarrow V : r_i \xleftarrow{\$} \mathbb{Z}_\lambda, \quad r \in PRIME \text{ } (\textit{same security parameters as discussed above})$$
$$P \to V : \pi \leftarrow x^{\left\lfloor \frac{2^T}{r} \right\rfloor} \pmod N$$
$$P, V : y \stackrel{?}{=} \pi^r x^{residue} \text{ } (\textit{should result in } x^{2^T}), \quad residue = 2^T \pmod r$$

Clearly computing π is not straightforward, for the same reason one cannot quickly compute $x^{2^T} \pmod N$ without the group order. Thus, the prover is forced to expend almost as many resources as it takes to compute y itself, to try and compute a division of the exponent. The prover uses the **one-the-fly long-division** algorithm to compute it in O(2T):

$$\pi \leftarrow 1 \in \mathbb{G} \tag{0.1}$$
$$residue \leftarrow 1 \in \mathbb{Z} \tag{0.2}$$
$$for \text{ } 0...T : \tag{0.3}$$
$$bit \leftarrow \left\lfloor \frac{2 \cdot residue}{r} \right\rfloor \in \{0, 1\} \tag{0.4}$$
$$residue \leftarrow 2 \cdot residue \pmod r \in \mathbb{Z}_\diagdown \tag{0.5}$$
$$\pi \leftarrow \pi^2 x^{bit} \in \mathbb{G} \tag{0.6}$$
$$output \text{ } \pi \tag{0.7}$$

It is important to note that this process is still extremely fast for the Trapdoor owner:

$$residue \leftarrow 2^T \pmod{r} \tag{0.8}$$

$$exponent \leftarrow (2^T \pmod{\phi(N)} - residue) \cdot r^{-1} \pmod{\phi(N)} \tag{0.9}$$

$$\pi \leftarrow x^{exponent} \pmod{N} \tag{0.10}$$

Comparison: there happen to be some important differences between these two approaches, which made me opt for [Wes18] for my blockchain contract:

1. *Proof Computation*:

   [Piet18] is a factor of $log(T)$ faster than [Wes18], which is only somewhat noticeable for practical circumstances on a normal machine. Anyway, this can be accounted for when choosing the VDF's T time parameter.

2. *Proof Verification*:

   [Wes18] has a much faster verification time: only 2 modular exponentiations; by comparison, [Piet18] has the same amount of exponentiations increased by a factor $log(T)$. This is less desirable for Ethereum contracts, where computations are very costly and the verification step will be performed on the blockchain.

3. *Proof Size*:

   [Wes18] has one group element, [Piet18] has $log(T)$. While this may not seem like too much, it would be a big price to pay for the blockchain ledger to store so many values per VDF transaction. This was the determining factor for me.

Non-interactive Proof Schemes

The previous proof schemes can be easily extended to a non-interactive variant by applying the Fiat-Shamir heuristic, *i.e. the challenge becomes a hash of the public parameters and proof*, under the Public Coin assumption. This process also guarantees proof **uniqueness** for a specific output.

Unfortunately, the [Wes18] proof evaluation and verification steps actually require the generation of a prime number, which can be achieved by checking whether the hash output is actually prime. If it is not, one can either keep generating more candidates using the hash with a counter, or just by testing all successive odd numbers for primality. For my implementation, I opted to use the probabilistic primality test Miller-Rabin, coupled with a deterministic test for the first 256 primes (in order to speed up computations on the blockchain).

Known Flaws and Future Improvements

- *Post-Quantum Safety:*

  the [RSW96] scheme, as well as all others based on exponentiation under unknown RSA group orders, is not safe against quantum computer attacks. Specifically, Shor's Quantum Factoring Algorithm is capable of factorizing the modulus and extracting the group order in polynomial time. Finding a post-quantum safe alternative is very much a new field of research, and current solutions suggest using incremental proofs (the structure is similar to [Piet18]'s reasoning) for proving hash chains, but there are no satisfactory results yet.

- *Time parameter Precision:*

  there is no current precise suggestion for good delay metrics. In research papers, $T = 2^{40}$ is a commonly chosen value. On my machine, 1 second was approximately $T = 2^{17}$, which would make $T = 2^{40} = 2^{40-17} seconds \approx 97 days$. There needs to be an independent analysis of just how long a single modular squaring takes, at best, on the fastest processors. This would allow us to reliably constrain all machines in the world to spend **at least** a certain amount of time on a computation. Of course, the fastest processors would still reach this computation before the others, so the revealed value shouldn't be used to reward on a first-come first-server basis. Currently, the fastest processors in the world seem to run at around 9Ghz, i.e. at most only 10 times faster than the popular Raspberry Pi Zero.

2) The Python Client

Requirements

For this project, I chose to implement the client in the latest **Python3.7**. The client is basically the program which allows offline evaluation for the VDF (would be too expensive on the blockchain), but it can easily be extended to become a program which automatically fetches new data from the Ethereum contract and interacts with it.

For convenience, the following standard libraries were used:

1. **random**: used to generate pseudo-random values for the probabilistic primality tests, and for the RSA setup.
2. **timeit**: used to fine-tune the VDF delay parameter for my machine.

For security purposes, I always try to keep 3rd party libraries to a minimum (especially considering today's incredible diffusion of flawed plugins, which can also bring forth new security vulnerabilities). However, I made a couple of reliable exceptions:

1. **gmpy2**: used to calculate the inverse of an element of a group. This library can be avoided simply by manually implementing the Extended Euclidean Algorithm, not at all a daunting task. However, i felt that this was not required for this academic proof of concept, since `gmpy2` is moderately popular on Python's distribution network PyPi, and is itself a wrapper around the very well known GNU Multiple Precision Arithmetic Library (GMP), which has been around for a long time.
2. **PyCrypto**: The Python Cryptography Toolkit is an incredibly well known implementation of famous cryptographic primitives for Python. I used it to provide a working secure hash function: Keccak256. This is because the EVM (Ethereum Virtual Machine) has its own internal and efficient implementation of Keccak256. While this function can be manually implemented in Python3, I wouldn't recommend it due to the risk of introducing flaws, as well as the required effort. Alternative implementations would be the C Openssl library (worldwide famous), other operating system libraries, the Web3 Python implementation of Solidity's common utilities (including the official keccak256 used on the contract), or hardware implementations from trusted vendors. It should be noted that Keccak256 is not the same thing as SHA3_256, which uses a slightly modified internal padding pattern, is not used by Ethereum, and was standardized by NIST in the U.S.A.

Program Structure

The Python script is implemented as a library, split into classes. It should be incredibly simple to just include this library and make calls to the functions found within, which are stateless (to preserve the same interface found within academic papers).

The following classes are present:

1. `VDF`: this class holds all the required functions to implement both the simple VDF from [RSW96], and the provable variant from [Wes18].
    - `TrapSetup: (Lambda, Time) -> (PublicParameters, Trap)`: implements the basic RSA VDF from [RSW96] and returns the group's secret and public parameters. Clearly, the secret key can be used as a trapdoor for the VDF: the `Trap` output contains the Totient of N. The `Lambda` input specifies the security for the underlying difficult problems (RSA Factoring). One side note: the difficulty for RSA Factoring is **not** derived from Lambda, but λ_{RSA} is Lambda itself... i know it might be a little confusing from an academic perspective.
    - `Eval: (PublicParameters, x) -> y`: evaluates a VDF problem from [RSW96].
    - `TrapEval: (PublicParameters, Trap, x) -> y`: same as `Eval`, but can use the trapdoor to quickly bypass the sequentiality restriction of VDFs.
    - `Eval_Wes18: (PublicParameters, x) -> (y, pi)`: implementation of the VDF evaluation with proof from [Wes18].
    - `TrapEval_Wes18: (PublicParametes, Trap, x) -> (y, pi)`: same as `Eval_Wes18`, but using the trapdoor.
    - `Verify_Wes18: (PublicParameters, x, y, pi) -> {True, False}`: the proof verification function from [Wes18]. This is not required by the client for the project, but it is good to have it for offline checking.
    - `HPrime: (_N, _Time, _x, _y) -> p`: the function required to implement the FiatShamir heuristic for [Wes18]. It hashes the public parameters and the proof, and then produces a large (256bit) random prime number. It doesn't necessarily need to be implemeneted in any specific manner as long as the results are random primes, but must be equivalent to the Contract Server's implementation. Hence, they are implemented in the exact same manner.
2. `RSA`: this class implements the setup phase for RSA groups. The modulus of an RSA group represents security for our VDF, and is used to hold soundness of the VDF together.
    - `Setup: Lambda -> (SecretKey, PublicKey)`: extracts public and private keys for an RSA cryptosystem, as well as the Totient of the modulus. Lambda is the security parameter which represents the length of the Modulus in bits. The owner of the Contract Server can use this function to generate public parameters for the creation of the contract, but must take care to use the same a Lambda which is compatible with the contract (e.g. 256 bits in our implementation, see the Known Flaws section).
3. `Primes`: this class implements the prime number generation for the RSA setup, as well as the primality tests for the VDF HPrime function.
    - `FIRST_256_PRIMES`: a constant containing all the first 256 primes of the natural field. It's useful as a primality checking performance speedup.
    - `CheckProbablePrime: p -> {True, False}`: a check for primality. If `p` is one of the first 256 primes in the naturals, it is a deterministic check. Otherwise, it is a probabilistic check, which can be easily set to an extremely low error rate, with low performance hit.
    - `CheckProbablePrime: Lambda -> p`: this function takes a Lambda parameter describing the desired space from which to randomly select a prime, and then finds one by alternating between generating a random value and using `CheckProbablePrime`.
    - `Check_MillerRabin: (p, tests) -> {True, False}`: a function used for probabilistic primality testing by `CheckProbablePrime`. Its error rate is $\frac{1}{4^{tests}}$. Originally I used $tests = 256$, but values such as 10 or 20 are still very effective, and much more suited to the performance constraints of Ethereum's Virtual Machine.

Within the script a function called `performance` can be used to fine tune the VDF timing parameter T, or to just check whether the trapdoor variants work as expected.

Finally, a commented out section at the bottom of the script provides the commands I used most often to help me manually interact with the contract.

Tools used

- Vim: a simple text editor sufficed for coding in python.
- IPython: this extremely versatile interactive Python3 console from the Jupyter team allows for quick editing and evaluation of small code snippets. I consider it invaluable for quick scripting.

Known Flaws and Future Improvements

- *Program Interactivity:*

  currently, this client acts as a library. It could be improved using cli/TUI/GUI toolkits to turn it into a user-friendly interactive program without much effort. While this is not required for this specific academic project, it would be essential for distributing a popular program.

- *Ethereum Connectivity:*

  currently, this program works standalone. Which means that, while being a "client", it is the user which makes the connection between parameters on the Contract and functions in the Script. Using some 3rd party library would allow the client to automatically connect to the blockchain, retrieve statistics and live information from the "server" contract, and offer the user interaction with the lottery system. Some secure mechanism would have to be devised to allow the client to carry out actions on the blockchain, by receiving enabling signatures from the user *without* revealing his secret credentials.

- *Automated Testing:*

  currently, completeness and soundness have been tested manually. While I am mostly satisfied with the fulfillment of these properties, given that Python3 is not a language with strong static typing (which, still, would not suffice to guarantee soundness): it would be better to have an automated system test them whenever the code is modified.

- *Improved Proof Evaluation Performance:*

  [Wes18] suggests an improved evaluation algorithm, windowed on-the-fly long-division, while [BBF18] suggests a parallelized variant for the one-the-fly long-division algorithm. These two techniques could be combined to offer noticeably improved proof evaluation times, but the efforts of doing that for this academic proof of concept are not justified.

- *Deterministic Primality Tests:*

  currently, primality is mainly checked using Miller-Rabin. An alternative agorithm, such as the AKS algorithm, would make this test deterministic. While this would not lead to a performance improvement for the client (unless the PRNG is switched to a much slower but more secure TRNG), it could be used as a reference for greatly improving performance on the blockchain, which suffers from the extraction of random values using public parameters and repeated hashing.

- *Bug:*

  the [Wes18] implementation requires working under $\mathbb{G} \setminus \{+1, -1\}$, this is not currently implemented.

3) The Solidity Server

Requirements

The server is implemented using the latest **Solidity0.5.4**. The language is still not incredibly mature and is missing quite a few features (such as big number exponentiation) from the EVM, but is at least well supported by the Ethereum developer ecosystem. The server's job is to **guarantee** correct functioning (and trustlessness, once the trapdoor is removed) of a lottery system to the outside world, through the blockchain's PoW mechanism. It also offers a publicly accessible API which can be used by other users on Ethereum to interact with the lottery, allowing them to play and automatically receive funds once the lottery is closed.

This smart contract is standalone, implemented only using functions present in the language core. Interactivity with the outside world is already provided by the Etherum blockchain once the contract (compiled for the EVM bytecode) is published.

To publish and test the contract for flaws, I had the opportunity to pick an official Ethereum test blockchain implementation, which allowed me to interact with the blockchain without using real money. I picked the testchain which was (at the time) fastest in processing transactions amongst those provided by Metamask: the "Kovan Test Newtowrk". Afterwards, an official testchain "faucet" dispensed free ETH (ether, the main currency unit used for Ethereum's PoW) upon request, and after having verified my identity through Github authentication.

Contract Structure

The contract is implemented as a single Smart Contract instance, split into private and public (i.e. API) functions, as well as the contract state.

The following **public state** is present:

1. *Contract Indexing:* some common variables are typically used to describe a contract to the outside world.
   - `name`: "TrapLottery", the project's name.
   - `symbol`: "LOT", sometimes used to index contracts and standardized "token" contracts.
   - `description`: used as a form of greeting.
2. *Contract Logic:* mappings and variables used to determine lottery mechanisms.
   - `PlayerLimit`: the amount of players allowed at any point in time. It is also used to limit player lottery guesses, in order to guarantee a decent payout to players of any round.
   - `authorized: address -> {true, false}`: commonly used to close off some contract functions to specific users. I use it to prevent people from funding a contract containing a trapdoor VDF, as well as allowing only the owner to start lottery rounds.
   - `players`: the users which are playing in the current open round, or have played in the previous closed one.
   - `winners`: the users which won the last closed round.
   - `submissions`: the actual lottery guesses each user has performed for the current open round or previous closed round.
   - `LotteryDuration`: the duration of a single lottery round, during which users can guess a winning number. This should be strictly shorter (consider that VDFs are not very sensitive to seconds) than the VDF's timing delay, otherwise the soundness of the system collapses. For this academic project, the value is kept to a low amount.
   - `LotteryStatus`: a string representing whether the lottery has started or not. Until the winning guess is revealed, the lottery is not closed, but may not be accepting guesses anymore either.
   - `LotteryStart`: the UNIX timestamp for when the current open (or previous closed) lottery round started. It is an actual timestamp, but beware that timing is not very sensitive to seconds on the blockchain (since it is restricted by network communication and needs to be verified by everyone).
3. *VDF Logic:* public parameters for the [Wes18] VDF implementation used for the lottery's security.
   - `N`: the modulus for the underlying RSA group.
   - `T`: the timing delay constant used for this VDF.
   - `Input`: the input for a VDF evaluation, also known as x.
   - `RawOutput`: the output for a VDF evaluation (excluding the proof), also known as y.
   - `Output`: the lottery's winning output, it is equal to $y \pmod{\textit{PlayerLimit}}$ and is meant to represent a human-like winning result.
   - `Proof`: the proof generated by the VDF evaluation, also known as π.
   - `Challenge`: the challenge required by the FiatShamir transformation of the VDF, generated by `HPrime`.

The following **contract events**, used for EVM execution logging of messages, are present:

1. `LotteryCreated`: notifies that the contract has been created, and logs the official public parameters for the VDF, as well as the `PlayerLimit` for the lottery.
2. `LotteryOpened`: notifies that a new round of the lottery has started, logs the randomly selected input.
3. `Submitted`: notifies that a user made a guess, logs who it was and what the guess was.
4. `Revealed`: notifies that the lottery's winning guess has been found, logs the corresponding VDF parameters.
5. `LotteryClosed`: notifies that the lottery's round is closed, right after the `Revealed` event is emitted, and logs the winning players as well as their distance from the winning guess.
6. `AddedModerator`: notifies that a new moderator, which can interact with all functions protected by the `authorized` variable, has been added. Mostly for auditing purposes.

The following contract **public API functions**, used by the players to interact with the lottery, are present:

1. `constructor: (_N, _PlayerLimit) -> {}`: instantiates the contract with VDF public parameters and the lottery's `PlayerLimit`. Also, adds to the moderators' `authorized` variable the contract creator.
2. `fallback (): msg.value -> {}`: the fallback function for a Solidity contract, used to capture any unknown calls, as well as payments to the contract. Only moderators (including the contract owner) are allowed to add funds into the lottery contract, since it is a trapdoor lottery.

3. `authorize: new_moderator -> {}`: adds a new moderator to the contract's `authorized` variable.
4. `start: {} -> {}`: as long as enough funds are available to grant each winner a "wei" (smallest Ethereum currency unit), then, it starts a lottery round by generating a random input VDF using the previous block's hash.
5. `submit: Guess -> {}`: as long as not too many players have already guessed, lets a new player save its guess in `submissions`, as well as its name in `players`.
6. `reveal: (RawGuess, GuessProof) -> {}`: validates the current round's [Wes18] proposed VDF output and proof, and closes the round. The winning players are calculated based on the set of users who got closest to the winning guess (guess capped by `PlayerLimit`), and the round pot (`PlayerLimit`) is split amongst them.
7. `HPrime: (_N, _T, _x, _y) -> p` the equivalent counterpart to the client's `HPrime` function, used to generate a trustless "large" (only 256 bits, see the Known Flaws) random prime number for the FiatShamir variant of [Wes18]. It was put into the API as a commodity to allow anybody to validate their own results independently, since it can be challenging to perfectly replicate its functionality across different languages (should one wish to use a different client, for example).

The following **internal private functions** are used by the contract:

1. `absdiff: (a,b) -> c`: finds the absolute value of a subtraction between two unsigned integers in Solidity. Surprisingly Solidity, which only currently fully supports the integer number type, did not provide this out of the box.
2. `expmod: (Base, Exponent, Modulus) -> b`: implements the Square-and-Multiply algorithm to achieve efficient moudular exponentiation. While this is already implemented by the EVM, Solidity does not yet have internal functions to take advantage of it.
3. `FIRST_256_PRIMES`: the first 256 primes in the naturals, used for optimized primality testing, just like in the client.
4. `check_probable_prime: p -> {true, false}`: tests a candidate to check whether it is a prime. If the candidate is one of the first 256 primes in the naturals it is a deterministic test, otherwise, it is a probabilistic test employing Miller-Rabin.
5. `miller_rabin: (p, tests) -> {true, false}`: probabilistic primality test used by `check_probable_prime`. Its error rate is the same as in the client's, but its performance is restriced due to the repeated hashing of the previous block's public variables (along with a counter) used to generate random values. Due to this, the number of tests is limited to 20, which is still a very good error rate.

Known Flaws and Future Improvements

- *BigInteger and Modular Arithmetic:*

  the lottery's implementation greatly suffers from Solidity's limitations. While the language seems simple enough to use, it falls short when used for traditional public-key cryptography. Not only does this limit usability, but it limits secuirity as well: all RSA group operations are capped to 256 bits, which is hardly secure against powerful attackers (2048 bits for the modulus are a current safe default). Implementations for integers of unlimited precision, as well as modular arithmetic (such as exponentiation) would be greatly appreciated. Third party libraries can probably be found for this, but I am skeptical on whether mature libraries for an immature language exist, and whether they are reliable from a security standpoint. It should be noted, however, that some of this is already implemented in the offical EVM, and could just be mapped to as a direct jump using some conversion between EVM bytecode and Solidity.

- *Deterministic Primality Tests:*

  the selection of random values through repeated hashing harms the primality tests' performance. Just as suggested for the client, the server should switch to a deterministic algorithm (e.g. AKS) in order to try and noticeably improve performance and reduce gas costs.

- *Transparency (no Trapdoor):*

  implementing a variant of [Wes18] which reliably generates groups of unknown order (e.g. class groups of imaginary quadratic fields) would lead to a completely trustless lottery contract, and is the natural progression for this project. Achieving this would allow turning lottery guesses into bids, where everybody can pitch into the pot, independently of the contract creator's funds (just like in real life). Rewards would either be equally distributed (due to a constant bidding amount), or would be based upon a percentage of how much a user bid on a pre-existing pot amount. The round's input selection would not need to be started by the contract moderators, but could be periodically generated as long as the pot reached a specific value.

- *Automated Testing:*

  manually testing for completeness and soundness on the blockchain is quite painful. Every action, including updating the contract, requires a transaction. The longer the transaction costs, the longer I typically had to wait for the result. In total, it can take up to hours to validate a seemingly correct contract using this process. Fortunately, there are alternatives: manually testing the contract on a locally genearated blockchain (this impacts the truthfulness of the tests), or using Solidity's automated testing system, coupling it with a debugger such as Remix. This is also a natural progression of the current project.

- *Efficiency Improvements:*

  developing smart contracts for the Etherum blockchain calls for a lot of efforts in reducing computational complexity, and therefore "gas" billing per contract call, since every other node on the blockchain is also forced to execute your code. This contract wouldn't even be allowed to publish without optimizations enabled, since instantiating it exceeded the gas limit for a transaction. These limits might be reduced on the testchains, since money is freely available up to a certain extent, but on the normal chain a contract's caller would still need to pay exhorbitant fees just to interact with the unoptimized variant.

- *Realistic Timing Fine-Tuning:*

  currently, the duration of a lottery round is 1 minute, and the VDf is tuned to delay the result for 2 minutes (on my machine). These values should be chosen to reflect reality: multiple minutes (or hours) for a normal lottery round are a better unit to give everyone the time to make transactions (a block's timestamp is in seconds but is not precise to the second in reality), and twice that amount might be ideal for choosing the VDF delay (also to ensure block confirmation on the blockchain). Strictly speaking, however, the VDF delay just needs to be greater than the duration of the lottery round; this might be hard to achieve in practice!

- *Reduced Logging:*

  usage of data is paid on the blockchain. Should too many users start using the contract, there would be a flood of events just for each bidding/guess. While this could be a desired outcome, I doubt it is manageable for practical purposes.

- *Better Randomness:*

  randomness on the blockchain does not really exist. In fact, VDF applications include creating an actual Randomness Beacon, since biased inputs **chosen** within the VDF delay lead to unbiased outputs. In this project, I extracted randomness from the blockchain using the previous block's hash to instantiate both a randomized input for the VDF, as well as the Miller-Rabin primality test. While this may look like a sufficiently fresh random value for these purposes (as long as the VDF delay is larger than the time the previous block has existed), miners on the blockchain may actually re-order or filter transactions in order to influence the block's public parameters. While we can counter this by hashing these parameters, the miners can still brutforce this hashed values. Therefore, we always need to extract large enough values from these hashes (e.g. 256 bits) such that they cannot be bruteforced, and use them in such a way that each bit of the value leads to a different result *(i.e. no use in generating 256 bit values and then chooosing the last bit, or a xor of all the bits, for the program logic)*. Furthermore, the current implementation of the contract only uses public block parameters to generate these pseudo-random values. It would be much better, in terms of security, to also use blockchain public parameters (e.g. gensis block's hash), as well as the contract's up-to-date parameters (e.g. VDF Modulus, contract address, etc). This prevents attackers/miners from abusing our randomness source, by tying the outcome directly to our current state.

- *Bug:*

  the [Wes18] implementation requires working under $\mathbb{G} \setminus \{+1, -1\}$, this is not currently implemented.

Tools Used

- Metamask: this popular browser extension makes it extremely easy to generate accounts for Ethereum (and its official testchains). It can also interact with website requests, allowing users to send transactions through a webapp. For a realistic scenario, with real money, a well audited secure hardware enclave (e.g. a secure PUF) would be a better approach to handling cryptocurrency accounts. For this academic project, real money is irrelevant.
- Etherscan: a good Ethereum blockchain explorer. It can be used to check out blockchain statistics, validated transactions, as well as interacting with published contracts through Metamask. Successful transactions are commonly referenced by people through an Etherscan link.
- Remix: a good IDE for developing and publishing Ethereum contracts. It can automatically compile Solidity contracts and publish them through Metamask.
- Kovan faucet: this website provides free ETH coins (up to a certain limit) at https://faucet.kovan.network.

Notes

The devlopment, testing and publishing process for the smart contract was far from ideal. Here are a few notes on that:

- Solidity0.5.4: while this language is good for simpler contracts, I had a really hard time making some things work. I would suggest trying out more mature languages, with more comprehensive and higher-level libraries, for compiling to EVM bytecode.
- Remix: this IDE is fantastic for on-demand coding in Solidity. However, it still needs some polishing. For example, testing code snippets was not immediately available. Publishing contracts also had some issues, as I always needed to re-instantiate the constructor variables or had to copy-and-paste the source code to Etherscan because automated source code publishing was not available.
- Etherscan: this is a really good explorer, but presentation of contracts is not fantastic. Visibility for public variables is certainly confusing, and event logs are formatted using raw hexadecimal values, not their natural type representation. Furthermore, the process for publishing a contract's source code can be problematic, including an annoying Google Captcha check as well as the fact that constructor values were not parsed correctly from the contract's bytecode (a separate website helped me fix this, https://abi.hashex.org/).