Ñíguez Randomity Engine

Scheich R. Ahmed
University of Westminster, London, United Kingdom

Introduction

Generating a pseudorandom number from the blockchain as an entropic source is a formidable task, especially when the miners influence the block gas limit, block-hash and block time, this may change as the network adheres to Proof of Stake (Casper POS). It depicts a stern complication for programmers who endure establishing a trust-less internal deterministic source of entropy for Ethereum blockchain, whose essence inclines to be transparent.

The complication

Since the blockchain is conspicuous, EVM bytecode on the blockchain is evident to all the existing nodes for

verification. Nevertheless, reverse-engineering (decompile) bytecode back to perfect original source code in a stack-based architecture is computationally impossible due to name omission of variables, functions and types; particularly if the smart contract is optimised, obfuscated and self-modifying. This settles one issue and generates another as there is a perception of reluctance and lack of confidence to interact with a closed source smart contract.

The solution

The block variables in their raw form are susceptible to influence or effortless prediction (In POW or Future Casper POS) by various parties. The solution to this problem is extensively mixed hashing (keccak256) of the contributing factors to the point that influencing block variables and predictions prove ineffective on the final hash output.

The problem here is after all the mixed hashing the creator of such smart contract will identify what the outcomes of certain blocks are and hence may use this to their advantage. Therefore, no entity will use such a facility as it is imperative and necessary to provide a degree of freedom and customisations to the random number end users.

The pseudorandom numbers generated are 24 sequences of unsigned 256 bits. The end user could use any sequence, cross sequence, allocate a slot in a

sequence and re-hash it. Thus, the smart contract creator has no influence due to the fact that the contract is immutable. Accordingly, the smart contract creator is incapacitated to comprehend and influence the sequences or slots of the random number a user picked. Consequently, a user can only have control over the customised sequence they generate but not the outcome of the random number.

Ñíguez Randomity Engine

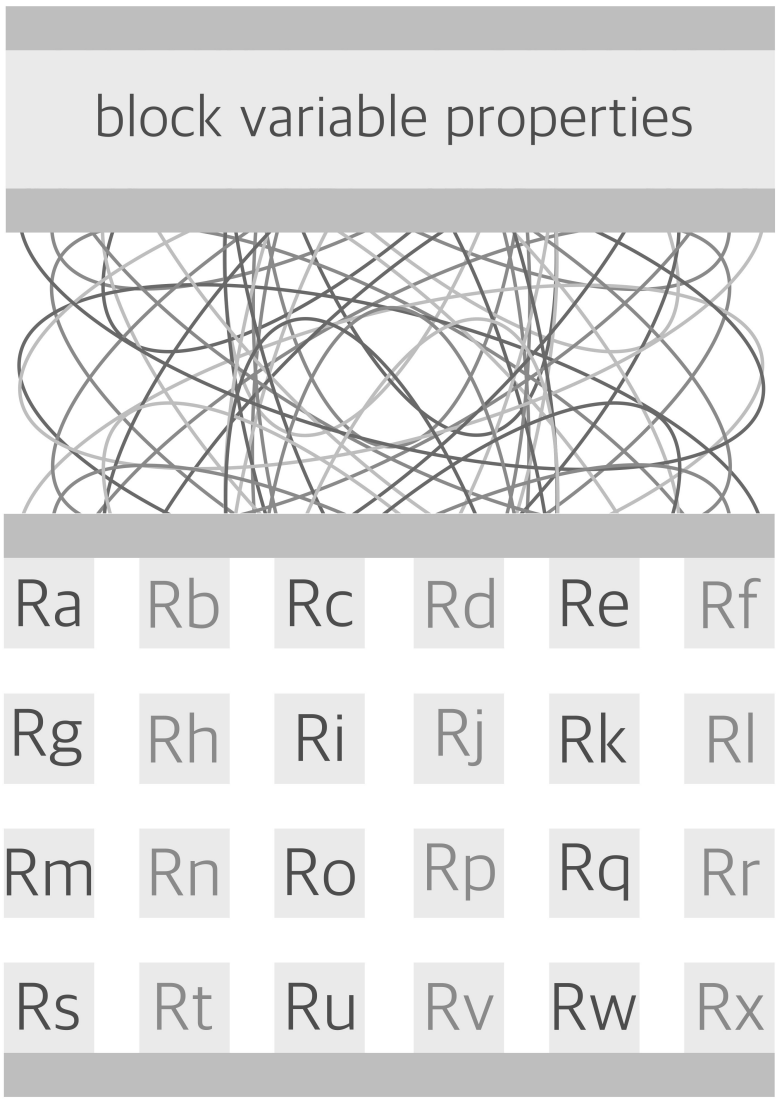
Ñíguez Randomity Engine is a smart contract deployed on the Ethereum blockchain that functions on the foretold protocol. It is immutable, providing impulsive sequences of pseudorandom number with every block.

In conventional pseudorandom number, the pattern tends to repeat, that is not the instance with this randomity engine as it uses a cryptographic hash function. Hence, every sequence is non-repeating and unique; if the sequences repeat, that would be considered an instance of a hash collision.

The engine still renders the sequence of random number even if the block variables are unpublished. Utilities, the likes of 'web3.eth.getStorageAt()' are redundant because nothing is stored in the storage and the contract dependencies are self-modifying.

The pseudorandom sequences generated are named alphabetically from 'Ra' to 'Rx', that is 1 to 24 respectively.

The illustration below elucidates the mechanism of this randomity engine.



Block variables are hashed, rehashed and mixed to generate 24 sequences of (256bits) unsigned integer.

Usage

The pseudorandom sequences generated can be used with infinite possibilities, a simple example is demonstrated here.

For the purpose of brevity, the use case has been limited here to nine mixed sequences of nine slots each.

Ra	Rb	Rc	Rd	Rl	Ro	Rp	Rt	Rx
6	2	6	6	3	5	8	3	2
3	6	3	3	2	1	7	5	5
2	8	4	4	6	6	7	7	8
5	8	5	5	1	1	9	4	7
1	6	7	5	9	0	9	7	8
4	5	7	2	1	6	2	8	1
2	4	1	3	4	5	8	9	5
2	2	0	7	2	3	4	0	1
7	2	3	5	1	3	7	8	8

Allocating slots for various pseudorandom sequences

((Ra()÷(10^3))%10)*(10^9) = 4000000000
((Rb()÷(10^6))%10)*(10^8) = 800000000
((Rc()÷(10^5))%10)*(10^7) = 50000000
((Rd()÷(10^2))%10)*(10^6) = 3000000
((Ro()÷(10^5))%10)*(10^5) = 100000
((Rd()÷(10^4))%10)*(10^4) = 50000
((Rp()%10)*(10^3)) = 7000
((Rt()÷(10^2))%10)*(10^2) = 900
((Rx()÷(10^8))%10)*(10) = 20
Add to complete sequence = 4853157920

This is purely an instance on how to set customised slots for sequences. If the classification is static then the numbers will be picked from the identical slots perpetually, hence it is advised to use dynamic slots by the user to be assured that the randomity engine creator is debilitated to predict the slots beforehand.

Applications

Econometric simulations and sampling

The significant motive to develop this randomity engine is to facilitate econometric analysis, run economic simulations and create artificial scenarios for statistical sampling while working with various heteroskedasticity models.

Monetary reward transactions

Conceivably the recommended application involving monetary reward is to set the slots for a future block number 'X' which is anonymous to anyone at the point of sending setting transaction. Nevertheless, if the future block number 'X; is influenced it will not make any difference and only after that future block variable is apparent the winner claims the price, this is a safe and secure practice.

Hash collision instances

Provided the robustness of Keccak256 towards collision resistance, it is only capable of generating 2^256 (that's a lot, current computation power will need more

time than the age of the universe to compute all hashes) hash outputs. However, in practice inputs are infinite; the pigeonhole principle assures that two different inputs, beyond 2^256, will unquestionably have identical hash outputs. The randomity engine is potentially used to identify such coincidental instances by archiving and matching.

Other Applications

Sophisticated ballot systems; where a voting individual has multiple favourable candidates but according to the law is required to cast one vote could make use of the randomity engine.

In Artificial intelligence for complex machine learning on the blockchain, the use of random number is crucial. Other applications include establishing artificial volatility which is necessary to discourage short-term asset speculation which is something being worked at and is in the mid-development stage. Another implementation is the random selection of validators in (PoS) Proof of Stake network.

MIT Licence

Ñíguez Randomity Engine is free to use by any entity for personal or commercial purpose and to explore the boundaries of the system. However certain state modifying transactions will consume gas.

Contract address for all networks:

0x031eaE8a8105217ab64359D4361022d0947f4572

MAINET | KOVAN | ROPSTEN | RINKEBY

Website:

niguezrandomityengine.github.io

Ethereum API:

API on GitHub

Guide:

Guide on How to use API