Ethereum Classic Technical Reference

Release 0.1

Christian Seberino

May 23, 2020

Contents

(The ETC Classic Technical Reference is still a work in progress. Feel free to send suggestions, feedback and corrections to me, Chris Seberino, at cs@etcplanet.org or through the Github pull request mechanism.)

Part I

Introduction

Ethereum Classic (ETC) is the most exciting technology today. It promises to upend governments, the financial industry, supply chain management and much more. The marriage of ETC with the Internet of Things is a game changer. Some think ETC may replace the World Wide Web!

ETC is composed of the world computer and database. The ETC world computer is also referred to as the ETC virtual machine. The ETC world database is also referred to as the ETC blockchain. These two

components have several noteworthy properties:

censorship resistance It is virtually impossible to stop the execution of code or to deny access to information.

security Security is built in with cryptography.

pseudonymity Users are anonymous except for pseudonyms.

openness All the software is open source, and, all the activity for all time is available for inspection.

reliability It is always available and virtually impossible to shut down.

trustlessness There is no need to rely on any single person or entity.

All of these properties, except trustlessness, are possible without blockchain technology. For example, consider a web server securely configured with several identical backup servers geographically distributed. Suppose these web servers were only accessible using onion routing. This setup can provide significant censorship resistance, security, pseudonymity, openness and reliability. Note however how much effort is required. With ETC, these properties are present by default!

Because ETC is trustless, *no one* has special powers. Therefore, it is possible to implement extremely sensitive applications on ETC such as financial and identity services.

Final note, everyone *effectively* runs their applications on the same single computer, the ETC world computer. However, this *virtual* computer is actually implemented by a worldwide network of computers. All the computers in the network run all applications in parallel. This extreme redundancy is a main reason the ETC world computer has its amazing properties.

Part II

World Computer (Virtual Machine)

The world computer is a single virtual machine implemented by a worldwide network of computers. Programs on the world computer are referred to as *smart contracts*. Because the world computer is Turing complete, smart contract can be arbitrarily complex. Users interact with the world computer through *clients* using *accounts*. In fact, *all* actions on the world computer are initiated by *transactions* sent from users.

1 Accounts

Accounts are associated with users and *smart contracts*. All accounts contain the following five components:

address These are sets of numbers used to identify accounts.

balance All funds are associated with accounts. This is a balance of classic ether, also know as ether or ETC.

code (smart contract) All smart contracts are associated with accounts. This component is an empty string for user accounts.

storage All smart contracts have associated memory. This component is an empty string for user accounts.

nonce Nonces are counters. For user accounts, these equal the number of associated *transactions*. For smart contract accounts, these equal the number of associated smart contracts created.

All components of all accounts together comprise the *state* of the world computer.

1.1 Addresses

All accounts are identified by addresses which are derived from secret random numbers unique to each account. These secret random numbers are referred to as *private keys*. Private keys must be kept private because they are used to digitally sign transactions from accounts. These transactions can transfer funds, create smart contracts, and, execute smart contracts. Strictly speaking, private key numbers must be between 1 and

```
115792089237316195423570985008687907852837564279074904382605163141518161494336
```

inclusive. This requirement is necessary for their use in ETC digital signatures. Some may be concerned that two users might unintentionally select the same private key. The odds of that happening are vanishingly small. In fact, the number of possible private keys is approximately equal to the number of atoms in the entire universe!

All private keys are associated 64 byte numbers derived from them which are referred to as *public keys*. The calculation of public keys involves an odd type of arithmetic with respect to pairs of numbers. Here is a Python script that calculates public keys from private keys:

```
#!/usr/bin/env python3

"""
Calculates ETC public keys from ETC private keys.

Usage: etc_pub_key <private key>
"""

import random
import sys

A           = 0
N           = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
P           = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
GX          = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
GY          = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
HEXADECIMAL = 16
NUM_FORMAT  = "{{:0{}x}}".format(len(hex(P)[2:]))

def inverse(number):
        """
        Inverts a number.
        """

        inverse = 1
        power   = number
        for e in bin(P - 2)[2:][::-1]:
                if int(e):
                        inverse = (inverse * power) % P
                power = (power ** 2) % P

        return inverse

def add(pair_1, pair_2):
        """
```

(continues on next page)

(continued from previous page)

```
        Adds two pairs.
        """

        if   pair_1 == "identity":
                sum_ = pair_2
        elif pair_2 == "identity":
                sum_ = pair_1
        else:
                if pair_1 == pair_2:
                        numer   = 3 * pair_1[0] ** 2 + A
                        lambda_ = (numer * inverse(2 * pair_1[1])) % P
                else:
                        numer   = pair_2[1] - pair_1[1]
                        denom   = pair_2[0] - pair_1[0]
                        lambda_ = (numer * inverse(denom)) % P
                x    = (lambda_ ** 2 - pair_1[0] - pair_2[0])  % P
                y    = (lambda_ * (pair_1[0] - x) - pair_1[1]) % P
                sum_ = (x, y)

        return sum_

def multiply(number, pair):
        """
        Multiplies a pair by a number.
        """

        product = "identity"
        power   = pair[:]
        for e in bin(number)[2:][::-1]:
                if int(e):
                        product = add(power, product)
                power = add(power, power)

        return product

def convert(pair):
        """
        Converts pairs to numbers by concatenating the elements.
        """

        return int("".join([NUM_FORMAT.format(e) for e in pair]), HEXADECIMAL)

        print(convert(multiply(int(sys.argv[1]), (GX, GY))))
```

The reason for this convoluted process is so that private keys cannot be derived from public keys. This allows public keys to be safely shared with anyone. If you want to learn more, investigate elliptic curve cryptography. The reason for this name is that historically it followed from calculations of the arc lengths of ellipses. Together, public and private keys are often referred to as *wallets*.

Addresses are formed from the first 20 bytes of the Keccak 256 hashes of public keys. These are more often used to identify accounts rather than public keys. Interestingly, public keys cannot be determined solely from addresses. Here is a Python script that calculates addresses from public keys. It requires the PySHA3 package. Addresses are typically expressed in hexadecimal notation and that convention is followed in this

script:

```
#!/usr/bin/env python3

"""
Calculates ETC addresses from ETC public keys.

Usage: etc_address <public key>
"""

import sha3
import binascii
import sys

N_ADDRESS_BYTES = 20
N_PUB_KEY_BYTES = 64

public_key = (int(sys.argv[1])).to_bytes(N_PUB_KEY_BYTES, byteorder = "big")
print(sha3.keccak_256(public_key).hexdigest()[-2 * N_ADDRESS_BYTES:])
```

Here is a slightly edited session, on a Linux computer, that calculates a public key and address with regards to a random private key. It uses the aforementioned scripts saved in files called etc_pub_key and etc_address respectively:

```
% PRIVATE_KEY=
↪"9278825938121281244563817223484328216764623708721224968735859314556303551842 4"

% PUBLIC_KEY=`etc_pub_key $PRIVATE_KEY`

% ADDRESS=`etc_address $PUBLIC_KEY`

% echo $PRIVATE_KEY
9278825938121281244563817223484328216764623708721224968735859314556303551842 4

% echo $PUBLIC_KEY
9808854183897174607002157792089896992612613490844656534725423301978228163634425857099752732031947328803

% echo $ADDRESS
89b44e4d3c81ede05d0f5de8d1a68f754d73d997
```

1.2 Ether & Gas

To create and utilize smart contracts, user submit transactions. Transactions must pay for these services in classic ether. Classic ether can be obtained by purchasing it or by mining. The smallest denomination of classic ether used in the ETC system is 1 wei. One billion billion (10^{18}) weis equal a single classic ether. Due to the mining reward formula, the total supply of classic ether will never exceed 210.6 million tokens.

The cost of creating and executing all smart contracts is measured in a made up unit referred to as gas units. Users submit transactions that pay for gas units in terms of classic ether. Notice that while the price of classic ether fluctuates, the price of various services in terms of gas units does not. In transactions, user specify how much classic ether they are willing to pay per gas unit. For security reasons, the amount of gas that can be purchased and used by blocks is limited.

2 Smart Contracts

Smart contracts are autonomous software applications that manage agreements. Agreements may be trivial or extremely complex. An alternative equivalent term is software *agents*. Consider vending machines. They specify and enforce agreements to release various items for various payments. They do not require humans to operate. Vending machine are therefore examples of smart contracts. They notion of smart contracts was conceived by Nick Szabo and predates blockchain technology:

> "A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs."
>
> – Nick Szabo, 1994

ETC makes an excellent smart contract platform. ETC programs autonomously manage countless agreements in a secure, reliable and trustless manner. For this reason ETC programs are referred to as smart contracts.

ETC smart contracts can read and write to their own storage as well as invoking other smart contracts. In this way, smart contracts can work together to provide increasingly sophisticated services.

Some, like Nick Szabo, envision smart contracts streamlining voluntary contractual agreements and disrupting the legal profession. Clearly software is less prone to misunderstanding and ambiguity than spoken languages! Others see a future where complex smart contracts replace entire corporations. Such programs are referred to as *distributed autonomous enterprises (DAEs)*. For example, imagine a smart contract implementing a ride sharing service. The smart contract could bring riders and drivers together in an efficient flexible manner. Note that ETC smart contracts can not only make existing agreements more efficient, but, they can also make possible contracts which previously were not possible due to overhead costs. For example, in addition to assisting multinational corporations, ETC can help teenagers running a small business and people providing microservices to third world countries.

Because the ETC world computer is implemented by a network of computers, ETC smart contracts are also referred to as decentralized applications, or *dapps* for short.

2.1 Smart Contract Languages

Typically smart contracts are written in high level languages. The corresponding source code is compiled to the equivalent ETC virtual machine instructions. The most popular high level smart contract language is Solidity. There are also other possible choices such as Vyper. Solidity is a Javascript like language designed to be easily adopted by new developers. Here is the Solidity source code for a simple program. All it does is maintain a counter variable. The counter can be incremented by anyone. Only the user account that created the smart contract can reset the counter value:

```
pragma solidity ^0.4.18;

/*
This smart contract maintains a counter which anyone can increment but only
the author can set to an arbitrary value.
*/

contract Counter {
        uint    counter;
        address author;
```

(continues on next page)

(continued from previous page)

```
        function Counter() public {
                counter = 0;
                author  = msg.sender;
        }

        function increment() public {
                counter += 1;
        }

        function set(uint new_value) public {
                if (msg.sender == author) {
                        counter = new_value;
                }
        }

        function get_counter() public constant returns (uint) {
                return counter;
        }
}
```

Here is Solidity source code for a more complex program. This one implements a new token:

```
pragma solidity ^0.4.18;

/*
Implements ChrisCoin which adheres to the Ethereum Token Standard.
*/

contract ChrisCoin {
        string                                          name_;
        string                                          symbol_;
        uint                                            decimals_;
        uint                                            total_supply;
        mapping(address => uint)                        balance;
        mapping(address => mapping(address => uint)) approved;

        event Approve(address indexed  managed_add,
                      address indexed  manager_add,
                      uint             approv_amt);
        event Transfer(address indexed send_add,
                       address indexed receiv_add,
                       uint            trans_amt);

        function ChrisCoin() public {
                /*
                Sets the named constants and the initial balance(s).
                */

                name_               = "ChrisCoin";
                symbol_             = "CHRC";
                decimals_           = 18;
```

(continues on next page)

(continued from previous page)

```
        total_supply        = 21000000 * 10 ** decimals_;
        balance[msg.sender] = total_supply;
    }

    function name() public constant returns (string) {
        /*
        Returns the cryptocurrency name.
        */

        return name_;
    }

    function symbol() public constant returns (string) {
        /*
        Returns the exchange ticker symbol.
        */

        return symbol_;
    }

    function decimals() public constant returns (uint) {
        /*
        Returns the maximum number of subdivision decimal places.
        */

        return decimals_;
    }

    function balanceOf(address account_add) public constant returns (uint) {
        /*
        Returns account balances.
        */

        return balance[account_add];
    }

    function allowance(address managed_add,
                       address manager_add)
                       public constant returns (uint) {
        /*
        Returns approved amounts.
        */

        return approved[managed_add][manager_add];
    }

    function approve(address manager_add,
                     uint approv_amt)
                     public constant returns (bool) {
        /*
        Returns approved amounts.
        */
```

(continues on next page)

(continued from previous page)

```
        approved[msg.sender][manager_add] = approv_amt;
        Approve(msg.sender, manager_add, approv_amt);

        return true;
    }

    function valid(address send_add,
                   address receiv_add,
                   uint trans_amt)
                   public constant returns (bool) {
        /*
        Determines the validity of transfers.
        */

        bool valid_trans_amt  = trans_amt <= total_supply;
        bool suff_send_bal    = balance[send_add] >= trans_amt;
        uint receiv_bal       = balance[receiv_add] + trans_amt;
        bool valid_receiv_bal = receiv_bal <= total_supply;

        return valid_trans_amt && suff_send_bal && valid_receiv_bal;
    }

    function update_balance(address send_add,
                            address receiv_add,
                            uint    trans_amt)
                            private {
        /*
        Updates balance with regards to tranfers.
        */

        balance[send_add]   -= trans_amt;
        balance[receiv_add] += trans_amt;
    }

    function update_approved(address send_add, uint trans_amt) private {
        /*
        Updates approved with regards to tranfers.
        */

        approved[send_add][msg.sender] -= trans_amt;
    }

    function transfer(address receiv_add,
                      uint trans_amt)
                      public constant returns (bool) {
        /*
        Transfers funds between accounts.
        */

        bool result = false;
        if (valid(msg.sender, receiv_add, trans_amt)) {
```

(continues on next page)

(continued from previous page)

```
                        update_balance(msg.sender, receiv_add, trans_amt);
                        Transfer(msg.sender, receiv_add, trans_amt);
                        result = true;
                }

                return result;
        }

        function transferFrom(address send_add,
                              address receiv_add,
                              uint trans_amt)
                              public constant returns (bool) {
                /*
                Transfers funds between accounts.
                */

                bool result     = false;
                bool approv_amt = trans_amt <= approved[send_add][msg.sender];
                if (valid(send_add, receiv_add, trans_amt) && approv_amt) {
                        update_balance(send_add, receiv_add, trans_amt);
                        update_approved(send_add, trans_amt);
                        Transfer(send_add, receiv_add, trans_amt);
                        result = true;
                }

                return result;
        }
}
```

2.2 Multisig Smart Contracts

Multisig smart contracts will likely be the dominant smart contract type in the future. The security and other benefits are that compelling.

Malware, keyboard loggers and "man in the middle attacks" are just some of the ways passwords can be stolen. Therefore, many use multifactor authentication to increase security. For example, accessing a website from a laptop may require a password and approval from a smartphone.

Ethereum Classic (ETC) and other smart contract systems can also benefit from multifactor authentication. ETC users are associated with accounts. ETC account authentication involves digital signatures. Therefore, ETC smart contracts requiring multifactor authentication are referred to as multisig smart contracts.

One of the most common types of multisig smart contracts requires digital signatures from any two of three accounts. Here are some applications where this is useful:

Single Individuals Imagine always requiring a digital signature from a laptop based account and a smartphone based account. To protect against the loss of either device, store the information for the third account in a secured paper wallet.

Online Shopping (Trusted Escrow) When purchasing products and services online, imagine buyers placing funds in multisig smart contracts. Have buyers and sellers each control an associated account. Allow an arbiter to control the third associated account. Notice buyers and sellers can together release funds without the arbiter. In the event of disagreements notice the arbiters can, together with

buyers or sellers, release funds to the desired choices. Because the arbiter does not control any funds, this is referred to as trusted escrow.

Small Businesses Imagine a small business controlling one associated account. Imagine a separate inspection service company controlling the second associated account. All transactions must be approved by the inspection service. To protect against any issues with the accounts, store the information for the third associated account in a secured paper wallet.

Here are two more multisig smart contract types and applications:

Majority Rule Imagine all members of a group controlling separate associated accounts. Imagine requiring digital signatures from any majority of the accounts. This would implement a majority rule arrangement.

Unanimity Rule Imagine all members of a group controlling separate associated accounts. Imagine requiring digital signatures from all of the accounts. This would implement a unanimity rule arrangement.

There are currently no ETC multisig smart contract standards. However, open source templates are available - such as from the OpenZeppelin project.

There are several common scenarios where multisig smart contracts are useful and significantly increase security. Therefore, it is likely they will take over ETC and the world.

3 Clients

To interact with the ETC world computer requires communicating with a computer on the ETC network. It is relatively easy to set up a computer to become part of the network. This requires the installation of an implementation of the ETC communication protocols. Possible choices include Geth, Parity and Mantis. To use ETC, it is not necessary to set up a new computer on the network. Applications can simply request information from other network computers. Such applications are referred to as *light clients*. Whether using a light client or setting up a full network computer, users can communicate with the ETC network using Web 3.

3.1 Web 3

Web3 refers to a standard set of ETC application programming interfaces using the Javascript Object Notation Remote Procedue Call (JSON RPC) protocol. Web3 provides a convenient way to interact with ETC nodes and the ETC system. The name Web3 refers to the most ambitious goal for Ethereum Classic (ETC) which is to replace the World Wide Web (Web). Blockchain based replacements for the Web are often referred to as Web 3.0.

The Web was developed by Tim Berners-Lee and first made publicly available in 1991. It is a user friendly general purpose system based on the Internet. Initially the Web mainly contained simple static content such as primitive personal home pages. As the Web evolved, greater dynamism and interactivity was possible such as with social media. This improved Web is often referred to as Web 2.0. The term was popularized by Tim O'Reilly.

Neither the Internet nor the Web were initially designed to be trustless systems. Components have been steadily introduced to improve security such as Transport Layer Security (TLS), certificate authorities, and, Domain Name System Security Extensions (DNSSEC). Unfortunately, many such improvements are only partially adopted.

Gavin Wood popularized the term Web 3.0 for blockchain based trustless alternatives to the Web. Confusingly, Web 3.0 also sometimes refers to the Semantic Web.

Web 3.0 is a peer to peer replacement for the Web. A peer to peer architecture is required to build trustless systems. Web 3.0 users are pseudonymous. They are only identified by their accounts, unlike the Web, where addresses can be associated with identities. ETC requires access to additional short and long term storage systems to replace the Web. The InterPlanetary File System (IPFS) is an example of a compelling peer to peer storage system that can integrate with ETC.

The Web currently coexists with blockchain systems. Websites access these systems to provide additional functionality. As ETC and related systems mature, browsers will increasingly just point to these Web alternatives thus ushering in the era of Web 3.0.

Part III

World Database (Blockchain)

The world database stores requests sent to the world computer. These requests are referred to as *transactions*. The transactions are collected into sets referred to as *blocks*. The blocks form a tree and a single path through that tree defines the *blockchain*. The blockchain stores other information in addition to transactions such as transaction *logs*. Lastly, process of creating, verifying and adding new blocks to the blockchain is referred to as *mining*.

The blockchain distributed database architecture was first introduced to the world in the Bitcoin system.

4 Transactions

Transactions are requests sent to the ETC network from user accounts. Transactions can send funds, create new smart contracts, or, execute existing smart contracts. Transaction resource requirements are measured in *gas* units. Gas is purchased with classic ether. All transactions contain the following six elements:

to (receiving address) Transactions contain receiving account addresses. This component is an empty string for smart contract creation transactions for which new accounts, with new addresses, will be created.

init or data (constructor or calling arguments) For smart contract creation transactions, this contains the associated constructors. For smart contract execution transactions, this contains the data operated on.

value (transfer amount) amount of classic ether, in units of wei, to be transferred to the receiving account

gas price offer of classic ether willing to pay per gas unit

gas limit (maximum gas purchase) maximum number of gas units willing to purchase

nonce originating user account nonces

v, r, s (digital signature) three numbers comprising the digital signature of the transaction with respect to the private key of the originating account

If applying transactions requires more gas to complete than the maximum gas amount allowed, then all the effects are reversed except that the user is still charged for the gas utilized.

5 Blocks

The ETC blockchain is composed of an array of blocks. Blocks contain three categories of information: *computation*, *consensus* and *context*. Blocks contain transaction related information (computation), mining related information (consensus), and, information to properly locate blocks on the blockchain (context). All components except for two lists form the block headers.

5.1 Computation

Transactions initiate all activity on the world computer. This category contains information related to this computation. Specifically, these block components consist of the following:

transaction list lists of transactions

transactions root (transaction list root hash) *root hashes* of transaction lists

gas used (transaction list total gas requirement) gas requirements for *all* the transactions in the transaction list

state root (transaction list final state root hash) *root hashes* of the states *after* each transaction is applied

receipts root (transaction log list root hash) *root hashes* of transaction log lists

logs Bloom (transaction log list Bloom filter) *Bloom filters* of transaction log lists

It may seem problematic that blocks only contain root hashes of states and transaction logs. Nevertheless, the full specification of any state or transaction log can always be obtained by reapplying all the transactions on the blockchain with respect to the initial state.

5.2 Consensus

Mining is the process of creating and validating new blocks. This is referred to as mining because the participants (miners) are rewarded with newly created ETC. The mining procedure is referred to as the consensus algorithm as it helps users of ETC agree on an ordered set of transactions. This involves a race to find certain numbers necessary to create new blocks. These numbers are referred to as *proof of work* information because they are "proof" that a certain amount of computational work was done. The block candidates that lose this race are referred to as the *uncle* blocks since they are related to the parents or last blocks added. These block components consist of the following:

extra data (miner extra data) 32 unused bytes added by miners

beneficiary (miner address) addresses with respect to block mining rewards

mix hash (miner validation help) values that help miners validate blocks faster

gas limit (miner gas maximum) maximum possible gas requirements to apply all transactions in blocks

nonce (proof of work information) the number required to add blocks to the blockchain

difficulty (proof of work difficulty) difficulty of finding proof of work information for the block

ommer header list (uncle header list) lists of the headers of the associated uncles

ommers hash (uncle header list root hash) Keccak 256 hashes of uncle header lists

The miner validation help components are necessary because slow block validation risks certain denial of service attacks. Miners are able to make slight adjustments to the miner gas maxima of the next blocks they create if desired. Uncles improve security by making attacks require performing more work. The consensus

algorithm automatically increases the proof of work difficulty for the next blocks when new blocks are being added too quickly. Likewise, the proof of work difficulty decreases when new blocks are being added too slowly.

5.3 Context

Blocks must always located correctly in the blockchain. Here are the blockchain components pertaining to context.

number (block number) the numbers of blocks that must precede blocks on the blockchain

parent hash (parent header hash) Keccak 256 hash of parent block headers

timestamp (date & time) dates and times that blocks were added to the blockchain

The parent block of a block is the preceding block on the blockchain. Dates and times are denoted by the number of seconds since 1970–01–01 00:00:00 UTC.

5.4 Accounts

There is no explicit account information in the blockchain. The only account information is the state root hash. To obtain account information, all the transactions in all the blocks of the blockchain must be implemented on the world computer with respect to the initial state.

6 Logs

For every transaction, the following information is logged:

- final state root hash
- cumulative gas usage

For example, the final state root hash logged with regards to the third transaction of block 5,889,421 is:

```
0x915dd6ca7dca0c1d68c3cc84e0d8551394f353042af35bd6b5cf21084d643a27
```

That is the state root hash after the first three transactions have been applied. Since all transactions for that block require 21,000 gas, the cumulative gas usage logged with regards to the third transaction is 63,000 gas. Smart contracts can request the logging of additional information.

6.1 Logging Requests

Smart contracts can request the logging of additional information. Specifically, smart contracts can request the logging of named lists of values. For example, suppose a smart contract based game wanted to record the following information for a player:

- account address
- health points
- gold coins

This information could be placed in a list named *Player* as in the following Solidity code declaration:

```
event Player(address user, uint256 health, uint256 gold);
```

Here is Solidity code to write specific player data to the blockchain:

```
emit Player(0xdf0b7310588741cad931de88bc6c4f687cdf0e16, 234, 198);
```

Note that variable values are stored but not variable names. Note also that user interfaces can access this logged information but that smart contracts cannot.

Logging requests are identified on the blockchain by hashes formed from their list names and list value types. For example, the aforementioned *Player* logging request is found on the blockchain by searching for the Keccak 256 hash of the string "Player(address,uint256,uint256)".

Bloom filters are always included with all smart contract logging requests. Bloom filters are hashes created from data to speed up searches with minimal storage requirements.

Indexed Values

Another way to speed up logging request searches is to store logging request values in a special way. Specifically, they can be stored as if they were additional logging request identifiers. This avoids having to extract them from the bytes encoding all the logging request values. Values stored in this manner are referred to as indexed values. For example, the Ethereum Token Standard specifies logging requests with indexed values as in this Solidity code:

```
event Transfer(address indexed sender, address indexed receiver, uint256 amount);
```

The log for transaction

```
0x104068d21afd428ce8eb5d9da155e11ba53414e40e088c884a678c6c203083d7
```

contains three logging request identifiers:

```
0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef

0x0000000000000000000000000efb32e82cf9d65a828d6d99e12f0beab01a467a6

0x0000000000000000000000000e71ac6142eaffc85ee3b9049facbcb13bc11402a
```

The first value is the Keccak 256 hash of the string "Transfer(address,address,unit256)". The other values are the sending and receiving addresses with regards to the corresponding token transfer.

7 Mining

Mining is the process of creating, validating, adding and distributing blocks. Computers, and their administrators, that perform this service are referred to as *miners*. Anyone computer on the network can become a miner. People are incentivised to become miners because of the financial rewards. Mining amazingly allows the Ethereum Classis system to be managed and secured in a trustless decentralized manner.

7.1 Proof Of Work Information

Valid blocks must contain certain numbers referred to as proof of work information. These numbers are also referred to as *nonces*. Miners compete to be the first to find this proof of work information and thereby

add new blocks to the blockchain. Finding proof of work information is intentionally made difficult. This difficulty is a main reason for the security of the blockchain.

The difficult process of finding adequate nonces involves the following. Nonces must be found such that certain hashes of the blocks, with the nonces added, have numerical values below specified maxima. The only way to find such hashes is to simply try as many nonce guesses as possible until adequate hashes are found. The maxima are automatically adjusted to keep the average block addition time around 15 seconds. Ethash is the hashing algorithm in this process.

7.2 Ethash

The Ethash hashing algorithm requires the determination of a certain extremely large directed acyclic graph that depends on block numbers. Quickly calculating several Ethash hashes requires storing the entire directed acyclic graph in memory. These large memory requirements thwart attempts to dominate the mining process by building application specific integrated circuits (ASICs).

7.3 Uncle Blocks

In the contest to add blocks to the blockchain, the losing blocks can be leveraged to increase the security of the blockchain. These losing blocks, to be used this way, must have parent blocks that are at most six blocks from the growing end of the blockchain. Miners gain additional financial rewards when they mention the hashes of the headers of these losing blocks in blocks that are accepted. This uncle block system is referred to as the GHOST protocol.

Here is why uncle blocks increase the security of the blockchain. The mining contest will inevitably create multiple chains of blocks. The convention is that the official chain is the one that is the most difficult to reproduce. Adding uncle blocks increases the difficulty of reproducing the official chain.

Uncle blocks are especially useful when blocks are not propagating quickly throughout the network. This leads to many losing blocks as miners keep adding blocks to outdated versions of the official chain. As block creation times thereby increase, the security of the network decreases. This is fortunately mitigated with uncle blocks.

7.4 Mining Pools

Because of the nature of the mining contest, the average expected mining rewards are proportional to the amount of computational resources dedicated to mining. There can still be variability in payout frequencies due to the random nature of the mining process. In order to deal with this variability, miners often join groups referred to as mining pools.

Some mining pools may lead to large amounts of mining resources in the control of a few individuals. Fortunately, there are trustless decentralized mining pools that avoid this risk.

7.5 Mining Rewards

Mining rewards consists of three parts:

Base Rewards This part depends on the block numbers. It is paid with newly created funds. Every five million blocks (about 2.4 years) this part decreases by 20%. Initially it was 5 ETC. It changed to 4 ETC after block number five million and will continue to change in the future.

Define the block era E as a function of the block number N as follows (// denotes integer division):

```
E = (N - 1) // 5000000
```

Then the base reward is as follows:

```
5  0.8:superscript:`E`
```

Uncle Rewards This part depends on the number of uncle blocks included as well as the block numbers. It is also paid with newly created funds. Each block can include at most two uncle blocks. The reward for each uncle block is an additional 3.125% of the base reward.

For the block era E and number of uncles U, the total uncle reward is as follows:

```
0.03125  U  (5  0.8:superscript:`E`)
```

After block number five million, miners that create the uncle blocks began getting this same reward per uncle block.

Gas Rewards This part depends on the transactions included. It is paid from the originating accounts. Miners execute the transactions and receive payments for the gas required. Each transactions specifies a price paid per unit gas.

For gas requirements G_1, G_2, G_3, … and corresponding gas prices P_1, P_2, P_3, …, the total gas reward is as follows:

```
G₁  P₁ + G₂  P₂ + G₃  P₃ + …
```

Therefore, the total reward for creating a block is the following:

```
(1 + 0.03125  U )  (5  0.8:superscript:`E`) + G₁  P₁ + G₂  P₂ + G₃  P₃ + …
```

Here is a Python script that uses this mining reward formula to calculate mining rewards:

```
#!/usr/bin/env python3

BASE_INITIAL  = 5
BASE_PERCENT  = 0.8
UNCLE_PERCENT = 0.03125
N_ERA_BLOCKS  = 5e6

def mining_reward(block_number, num_uncles, gas_reqs, gas_prices):
        """
        Calculates mining rewards from block information.  The gas
        information must be provided in lists or tuples.  The gas
        prices must be in ETC.
        """

        era           = (block_number - 1) // N_ERA_BLOCKS
        base_reward   = (BASE_PERCENT ** era) * BASE_INITIAL
        uncle_reward  = UNCLE_PERCENT * base_reward
        uncle_rewards = num_uncles * uncle_reward
        gas_rewards   = 0
        for (gas_req, gas_price) in zip(gas_reqs, gas_prices):
                gas_rewards += gas_req * gas_price

        return base_reward + uncle_rewards + gas_rewards
```

Here are some example calculations on real ETC blockchain data:

```
>>> mining_reward(5425392, 0, [], [])
4.0
>>> mining_reward(5423326, 1, [], [])
4.125
>>> mining_reward(5424471, 0, [36163, 36163] , [2e-8, 2e-8])
4.00144652
>>> mining_reward(5421363, 1, [21000, 21000, 21000, 21000, 21000], [5.5e-8, 2e-8, 2e-8,␣
↪1.6e-8, 1e-8])
4.127541
```

The mining reward formula bounds the supply of ETC. Notice only the base and uncle rewards increase the supply since the gas rewards just transfer existing funds. Because the uncle rewards vary, the eventual total supply of ETC can only be approximated.

The formula for the future increase in supply per era, assuming a constant number of uncle blocks, is the following:

```
5000000   (1 + 2   0.03125   U )   (5   0.8:superscript:`E`)
```

The factor of 2 is necessary to include the uncle block creator rewards. The total supply can be estimated from this formula by adding the contributions for the remaining eras. Era 192, which will occur around the year 2474, is the last era which increases the supply.

Assuming no more uncle blocks gives a lower bound of about 198.3 million ETC. Assuming the maximum number of uncle blocks gives an upper bound of about 210.6 million ETC.

Part IV

Appendices

blank

8 Recursive Length Prefix

Serialization is the process of encoding data structures into byte sequences. It is also referred to as marshalling and pickling. Serialization is necessary when storing and sending data structures.

RLP is a serialization format created by Ethereum developers for storage and communications. It is used for all data structures such as accounts, transactions and blocks. RLP is simpler than the alternatives such as Extensible Markup Language (XML), JavaScript Object Notation (JSON), Binary JSON (BSON), Protocol Buffers and Bencode.

RLP is also consistent. The same inputs are always converted to the same byte sequences. This is not true of all serialization formats. For example, when encoding sets of key value pairs, some schemes do not specify an ordering.

RLP operates on byte sequences and lists. Lists can contain byte sequences and other lists. The interpretation of all inputs is handled by other protocols. For byte sequences, small headers are added which depend on the length. For lists, the elements are encoded separately and concatenated. As with byte sequences, small headers are added which depend on the length. Lastly, all lengths are encoded in big endian format.

Here are Python functions which implement RLP encoding and decoding:

```
#!/usr/bin/env python3

import math

N_BITS_PER_BYTE = 8

def n_bytes(integer):
        """
        Finds the numbers of bytes needed to represent integers.
        """

        return math.ceil(integer.bit_length() / N_BITS_PER_BYTE)

def get_len(input, extra):
        """
        Finds the lengths of the longest inputs using the given extra values.
        """

        n_bytes = input[0] - extra

        return 1 + n_bytes + int.from_bytes(input[2:2 + n_bytes], "big")

def rlp_encode(input):
        """
        Recursive Length Prefix encodes inputs.
        """

        if isinstance(input, bytes):
                body = input
                if   (len(body) == 1) and (body[0] < 128):
                        header = bytes([])
                elif len(body) < 56:
                        header = bytes([len(body) + 128])
                else:
                        len_   = len(body)
                        len_   = len_.to_bytes(n_bytes(len_), "big")
                        header = bytes([len(len_) + 183]) + len_
                result = header + body
        else:
                body = bytes([])
                for e in input:
                        body += rlp_encode(e)
                if len(body) < 56:
                        header = bytes([len(body) + 192])
                else:
                        len_   = len(body)
                        len_   = len_.to_bytes(n_bytes(len_), "big")
                        header = bytes([len(len_) + 247]) + len_
                result = header + body

        return result
```

(continues on next page)

(continued from previous page)

```
def rlp_decode(input):
        """
        Recursive Length Prefix decodes inputs.
        """

        if   input[0] < 128:
                result = input
        elif input[0] < 184:
                result = input[1:]
        elif input[0] < 192:
                result = input[1 + (input[0] - 183):]
        else:
                result = []
                if input[0] < 248:
                        input = input[1:]
                else:
                        input = input[1 + (input[0] - 247):]
                while input:
                        if   input[0] < 128:
                                len_ = 1
                        elif input[0] < 184:
                                len_ = 1 + (input[0] - 128)
                        elif input[0] < 192:
                                len_ = get_len(input, 183)
                        elif input[0] < 248:
                                len_ = 1 + (input[0] - 192)
                        else:
                                len_ = get_len(input, 247)
                        result.append(rlp_decode(input[:len_]))
                        input = input[len_:]

        return result
```

Notice that the functions are recursive. Notice also that the functions work for inputs requiring up to about 18 million terabytes. Here are examples of their usage:

```
>>> rlp_encode(b"A")
b'A'

>>> rlp_encode(b"12345")
b'\x8512345'

>>> rlp_encode(20 * b"12345")
b
↪'\xb8d12345123451234512345123451234512345123451234512345123451234512345123451234512345123
  45123451234512345'

>>> rlp_encode([b"12345"])
b'\xc6\x8512345'

>>> rlp_encode([b"abcde", 3 * [b"12345"], [b"fghij"], b"67890", 4 * [b"klmno"]])
```

(continues on next page)

(continued from previous page)

```
b
↪'\xf8\x85abcde\xd2\x8512345\x8512345\x8512345\xc6\x85fghij\x8567890\xd8\x85klmno\x85klmno
  \x85klmno\x85klmno'

>>> rlp_decode(b"\x8512345")
b'12345'

>>> rlp_decode(b"\xc6\x8512345")
[b'12345']

>>> rlp_decode(b
↪"\xf8\x85abcde\xd2\x8512345\x8512345\x8512345\xc6\x85fghij\x8567890\xd8\x85klmno\x85klmno\x85klmno\x8
↪")
[b'abcde', [b'12345', b'12345', b'12345'], [b'fghij'], b'67890', [b'klmno', b'klmno', b
↪'klmno', b'klmno']]
```

RLP is an elegant and approachable serialization format used extensively by ETC. It can be quickly mastered thereby illuminating this important aspect of the system.

9 Root Hashes

The Ethereum Classic (ETC) blockchain contains "root hashes" that help maintain the integrity of various components of the ETC system. I will describe these root hashes including how to calculate them.

Some important ETC data structures are sets of key value pairs that are stored as Merkle Patricia tries. Tries are trees of nodes. The top nodes correspond to the "roots" of the trees. Therefore, hashes associated with the top nodes of Merkle Patricia tries are referred to as root hashes. Specifically, root hashes are the Keccak 256 hashes of the Recursive Length Prefix (RLP) encodings of the top nodes.

ETC block headers contain root hashes for states, transaction lists and receipt lists. ETC block headers also implicitly specify storage root hashes in the state root hashes.

Here is Python code that implements RLP encoding and decoding:

```
import math

BYTE_LEN = 8

def n_bytes(integer):
        """
        Finds the numbers of bytes needed to represent integers.
        """

        return math.ceil(integer.bit_length() / BYTE_LEN)

def get_len(input, extra):
        """
        Finds the lengths of the longest inputs using the given extra values.
        """

        n_bytes = input[0] - extra
```

(continues on next page)

(continued from previous page)

```
        return 1 + n_bytes + int.from_bytes(input[2:2 + n_bytes], "big")

def encode(input):
        """
        Recursive Length Prefix encodes inputs.
        """

        if isinstance(input, bytes):
                body = input
                if   (len(body) == 1) and (body[0] < 128):
                        header = bytes([])
                elif len(body) < 56:
                        header = bytes([len(body) + 128])
                else:
                        len_   = len(body)
                        len_   = len_.to_bytes(n_bytes(len_), "big")
                        header = bytes([len(len_) + 183]) + len_
                result = header + body
        else:
                body = bytes([])
                for e in input:
                        body += encode(e)
                if len(body) < 56:
                        header = bytes([len(body) + 192])
                else:
                        len_   = len(body)
                        len_   = len_.to_bytes(n_bytes(len_), "big")
                        header = bytes([len(len_) + 247]) + len_
                result = header + body

        return result

def decode(input):
        """
        Recursive Length Prefix decodes inputs.
        """

        if   input[0] < 128:
                result = input
        elif input[0] < 184:
                result = input[1:]
        elif input[0] < 192:
                result = input[1 + (input[0] - 183):]
        else:
                result = []
                if input[0] < 248:
                        input = input[1:]
                else:
                        input = input[1 + (input[0] - 247):]
                while input:
                        if   input[0] < 128:
                                len_ = 1
```

(continues on next page)

(continued from previous page)

```
                        elif input[0] < 184:
                                len_ = 1 + (input[0] - 128)
                        elif input[0] < 192:
                                len_ = get_len(input, 183)
                        elif input[0] < 248:
                                len_ = 1 + (input[0] - 192)
                        else:
                                len_ = get_len(input, 247)
                        result.append(decode(input[:len_]))
                        input = input[len_:]

        return result
```

Here is Python code that calculates root hashes using the PySHA3 package. It requires the RLP code above to be saved to an accessible location with the file name rlp.py. Invoke the root_hash function on Python dictionaries representing sets of ETC key value pairs. All keys and key values must be Python byte strings:

```
import sha3
import rlp

HASH_LEN = 32
HEXADEC  = 16

def remove(dict_, segment):
        """
        Removes initial key segments from the keys of dictionaries.
        """

        return {k[len(segment):] : v for k, v in dict_.items()}

def select(dict_, segment):
        """
        Selects dictionary elements with given initial key segments.
        """

        return {k : v for k, v in dict_.items() if k.startswith(segment)}

def find(dict_):
        """
        Finds common initial segments in the keys of dictionaries.
        """

        segment = ""
        for i in range(min([len(e) for e in dict_.keys()])):
                if len({e[i] for e in dict_.keys()}) > 1:
                        break
                segment += list(dict_.keys())[0][i]

        return segment

def patricia_r(dict_):
        """
```

(continues on next page)

(continued from previous page)

```
        Creates Patricia tries that begin with regular nodes.
        """

        pt = (HEXADEC + 1) * [None]
        if "" in dict_:
                pt[-1] = dict_[""]
                del(dict_[""])
        for e in {e[0] for e in dict_.keys()}:
                pt[int(e, HEXADEC)] = patricia(remove(select(dict_, e), e))

        return pt

def patricia_s(dict_):
        """
        Creates Patricia tries composed of one key ending special node.
        """

        pt = list(dict_.items())[0]
        if len(pt[0]) % 2 == 0:
                pt = (bytes.fromhex("20" + pt[0]), pt[1])
        else:
                pt = (bytes.fromhex("3"  + pt[0]), pt[1])

        return pt

def patricia(dict_):
        """
        Creates Patricia tries from dictionaries.
        """

        segment = find(dict_)
        if   len(dict_) == 1:
                pt = patricia_s(dict_)
        elif segment:
                dict_ = remove(dict_, segment)
                if len(segment) % 2 == 0:
                        pt = [bytes.fromhex("00" + segment), patricia_r(dict_)]
                else:
                        pt = [bytes.fromhex("1"  + segment), patricia_r(dict_)]
        else:
                pt = patricia_r(dict_)

        return pt

def merkle(element):
        """
        Encodes Patricia trie elements using Keccak 256 hashes and RLP.
        """

        if   not element:
                merkle_ = b""
        elif isinstance(element, str):
```

(continues on next page)

(continued from previous page)

```
                merkle_ = bytes.fromhex(element)
        elif isinstance(element, bytes):
                merkle_ = element
        else:
                merkle_ = [merkle(e) for e in element]
                rlp_    = rlp.encode(merkle_)
                if len(rlp_) >= HASH_LEN:
                        merkle_ = sha3.keccak_256(rlp_).digest()

        return merkle_

def merkle_patricia(dict_):
        """
        Creates Merkle Patricia tries from dictionaries.
        """

        return [merkle(e) for e in patricia(dict_)]

def root_hash(dict_):
        """
        Calculates root hashes of Merkle Patricia tries from dictionaries.
        """

        dict_ = {k.hex() : v for k, v in dict_.items()}

        return sha3.keccak_256(rlp.encode(merkle_patricia(dict_))).hexdigest()
```

Here are sample calculations for all of the root hash types found in the ETC blockchain. They require the root hash code above to be saved to an accessible location with the file name root_hash.py. The RLP code above must be saved to an accessible location with the file name rlp.py. Lastly, the following code to convert integers to Python byte strings must be saved to an accessible location with the file name int_to_bytes.py:

```
def int_to_bytes(number):
        if number:
                hex_ = hex(number)[2:]
                if len(hex_) % 2 != 0:
                        hex_ = "0" + hex_
                result = bytes.fromhex(hex_)
        else:
                result = b""

        return result
```

For state root hash calculations, the keys of the Python dictionaries must be the Keccak 256 hashes of the account addresses. The key values must be the RLP encodings of lists containing the corresponding account nonces, balances, storage root hashes, and, smart contract hashes. One way to obtain state information is with an ETC Geth node. For example, the following ETC Geth node command prints the state information for block 1,000,000:

geth dump 1000000

Here is the beginning of the voluminous output:

```
{
    "root": "0e066f3c2297a5cb300593052617d1bca5946f0caa0635fdb1b85ac7e5236f34",
    "accounts": {
        "843fd22c88d59e57ae1856a871a5d95e95b0a656": {
            "balance": "52500000000000",
            "nonce": 1,
            "root": "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash":
            "c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470"
              ,
            "code": "",
            "storage": {}
        },
        "dcd0b6fa4f0a26a7b12325b0d09b5b809c5aef84": {
            "balance": "9375377890126000",
            "nonce": 1,
            "root": "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash":
            "c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470"
              ,
            "code": "",
            "storage": {}
        },
        "7d62878a7235e95d56f802f80835543cac711f90": {
            "balance": "204544100000000000",
            "nonce": 0,
            "root": "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash":
            "c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470"
              ,
            "code": "",
            "storage": {}
        },
        "67db390312dc02a140c358add4f37966c7775096": {
            "balance": "0",
            "nonce": 2,
            "root": "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash":
            "c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470"
              ,
            "code": "",
            "storage": {}
        },

...etc.
```


The following code prints the state root hash for block 1,000,000 which is 0x0e066f3c2297a5cb300593052617d1bca5946f0caa0635fdb1b85ac7e5236f34. It requires the aforementioned state information to be saved to an accessible location with the file name state_1000000:

```
import root_hash
import sha3
import rlp
```

(continues on next page)

(continued from previous page)

```
import int_to_bytes

dict_ = {}
state = eval(open("state_1000000", "r").read())
for address in state["accounts"]:
        account    = state["accounts"][address]
        account    = [int_to_bytes.int_to_bytes(int(account["nonce"])),
                      int_to_bytes.int_to_bytes(int(account["balance"])),
                      bytes.fromhex(account["root"]),
                      bytes.fromhex(account["codeHash"])]
        key        = sha3.keccak_256(bytes.fromhex(address)).digest()
        value      = rlp.encode(account)
        dict_[key] = value

print(root_hash.root_hash(dict_))
```

For transaction list root hash calculations, the keys of the Python dictionaries must be the RLP encodings of the transaction indices starting from zero. The key values must be the RLP encodings of lists containing the corresponding transaction nonces, gas prices, gas usage maxima, destination addresses, ether sent, data sent and digital signature components. The following code prints the transaction list root hash for the transactions in block 4,000,003 which is 0xad79d498b7e407d3a2b32c13a380ee93635da2b3e0696c39563cbd5c32d368b2:

```
import root_hash
import sha3
import rlp
import int_to_bytes

key_1     = rlp.encode(int_to_bytes.int_to_bytes(0))

nonce     = int_to_bytes.int_to_bytes(1514565)
gas_price = int_to_bytes.int_to_bytes(20000000000)
gas_max   = int_to_bytes.int_to_bytes(50000)
dest      = 0x7b96a5006d5fc86d05f8799fe1fc6f7d23b24969
dest      = int_to_bytes.int_to_bytes(dest)
ether     = int_to_bytes.int_to_bytes(1001525814273650153)
data      = b""
v         = int_to_bytes.int_to_bytes(157)
r         = 0x8815ebbcdb56717a30193db4629fa7565d2fb06c6fba2aaf0db06deaf932955d
r         = int_to_bytes.int_to_bytes(r)
s         = 0x4dbd4dcb648114859f57122d804b85c2dd60d0b502fb93d0ef770d50bfa3a59d
s         = int_to_bytes.int_to_bytes(s)
trans     = [nonce, gas_price, gas_max, dest, ether, data, v, r, s]
value_1   = rlp.encode(trans)

key_2     = rlp.encode(int_to_bytes.int_to_bytes(1))

nonce     = int_to_bytes.int_to_bytes(43565)
gas_price = int_to_bytes.int_to_bytes(20000000000)
gas_max   = int_to_bytes.int_to_bytes(21000)
dest      = 0x7ccfb3028404225e4e9da860f85274e30ccc9275
dest      = int_to_bytes.int_to_bytes(dest)
ether     = int_to_bytes.int_to_bytes(109404508089999998976)
```

(continues on next page)

(continued from previous page)

```
data      = b""
v         = int_to_bytes.int_to_bytes(28)
r         = 0x8d6e2fcfe032d2612d2ea56da6d07b6a94004a4ec7cbe2c3f086db1a194aa679
r         = int_to_bytes.int_to_bytes(r)
s         = 0x6ed1333497c12b4549e55d117977bf60bb96872dfb05816fb7ce25c7396ef23a
s         = int_to_bytes.int_to_bytes(s)
trans     = [nonce, gas_price, gas_max, dest, ether, data, v, r, s]
value_2   = rlp.encode(trans)

print(root_hash.root_hash({key_1 : value_1, key_2 : value_2}))
```

For receipt list root hash calculations, the keys of the Python dictionaries must be the RLP encodings of the receipt indices starting from zero. The key values must be the RLP encodings of lists containing the corresponding receipt state root hashes, cumulative gas amounts, log Bloom filters and logs. The following code prints the receipt list root hash for the receipts in block 4,000,003 which is 0x4b3b43affc2927a152b9d6f18e378cf33671f8606e8549de292ae36b8a691584:

```
import root_hash
import sha3
import rlp
import int_to_bytes

key_1   = rlp.encode(int_to_bytes.int_to_bytes(0))

state   = "abca6dd8fb332962c1c14c02d13b2082aee152496dc809d9642e2deca07fb7c2"
gas     = 0x5208
bloom   = 256 * "00"
logs    = []
receipt = [bytes.fromhex(state),
           int_to_bytes.int_to_bytes(gas),
           bytes.fromhex(bloom),
           logs]
value_1 = rlp.encode(receipt)

key_2   = rlp.encode(int_to_bytes.int_to_bytes(1))

state   = "029b0eb2c76ff08a1cf47aba4be53ff1c20b01026206eca248b47e0657f97524"
gas     = 0xa410
bloom   = 256 * "00"
logs    = []
receipt = [bytes.fromhex(state),
           int_to_bytes.int_to_bytes(gas),
           bytes.fromhex(bloom),
           logs]
value_2 = rlp.encode(receipt)

print(root_hash.root_hash({key_1 : value_1, key_2 : value_2}))
```

For storage root hash calculations, the keys of the Python dictionaries must be the Keccak 256 hashes of the storage indices for all nonzero storage values. The key values must be the RLP encodings of the corresponding storage values. The following code prints the storage root hash for the account with the address 0xd4eae4ae8565f3ecf218191fb267941d98a2c77a which is 0x9f630ea9c8cc6e9f7ecbc08cb7f9e901c14b788cc8f2ae64e3134cf3cb089f55. Note that this result was correct

as of block 5,874,861 but may possibly change afterwards:

```
import root_hash
import sha3
import rlp
import int_to_bytes

KEY_LEN = 32
ZERO    = b"\x00"

dict_   = {}
storage = [(0, 0x51f24771a5a2720456076e7c81d59753dac20e1f),
           (1, 0x4563918244f40000),
           (3, 0x55c64da8),
           (4, 0x6f05b59d3b20000),
           (5, 0x4fb5acbe16ffdda225cb14c64aa84c7e253b08ae)]
for e in storage:
        key        = int_to_bytes.int_to_bytes(e[0])
        key        = (KEY_LEN - len(key)) * ZERO + key
        key        = sha3.keccak_256(key).digest()
        value      = rlp.encode(int_to_bytes.int_to_bytes(e[1]))
        dict_[key] = value

print(root_hash.root_hash(dict_))
```

Root hashes are vital for the operation of the ETC world computer. The ETC system utilizes state, transaction list, receipt list and storage root hashes. These ETC root hashes can be found with a detailed recipe involving RLP encodings, Keccak 256 hashes and Merkle Patricia tries.

10 Bloom Filters

Millions of people search the Internet, government databases, private databases and blockchains everyday for medical advice, financial updates, weather reports, maps and more. Likewise, millions of people want or need fast searches.

An effective method to speed up many searches is the use of indexes. Indexes, like those at back of textbooks, provide the locations of all search terms. A possible drawback is that they require large amounts of storage.

An effective method to speed up many searches, with less storage requirements, is the use of Bloom filters. These are important in many areas such as mobile and embedded devices.

Bloom filters are binary strings used to quickly determine set membership with nominal storage requirements. Browsers use them to detect malicious websites. Databases use them to avoid searching for nonexistent data.

Bloom filters require less memory than indexes, but, they sometimes give false positives. In other words, they might claim an object is a member of a set when it is not. It is noteworthy that Bloom filters never give false negatives. They never claim an object is not in a set when it actually is. Browser Bloom filters might incorrectly claim safe websites are malicious, but, they will never claim malicious websites are safe. Fortunately, extra checks can always be performed to eliminate any false positives.

To build a Bloom filter for the set {X_1, X_2, X_3, ..., X:subscript:n}, with hash function H, calculate H(X_1) | H(X_2) | H(X_3) | ... | H(X:subscript:n). | is the bitwise OR operator. H is only valid if the number of set bits (ones) in all hashes is always less than or equal to some selected maximum.

Larger Bloom filters have less false positives. Bloom filters of several megabytes are not uncommon. As hash functions typically do not have large enough hashes, not to mention hashes with required number of

set bits, adequate hashes are often constructed from multiple hash functions. A common recipe is to have a group of hash functions each set a single bit in the construction of a valid hash. For example, to set a bit in a hash with 2^{23} bits, the first 23 bits of the Secure Hash Algorithm 1 (SHA1) hash can be used to select the position of a bit to set.

To determine if an object X might be in a set with Bloom filter B, built with hash function H, determine if H(X) & B = H(X). & is the bitwise AND operator. Notice that the test only returns true if all the set bits in H(X) are also set in B. Basically, groups of set bits in B correspond to elements. X might be a member of the set if and only if its group of set bits corresponds to a group in B. The reason B can only determine if X might be in the set is because B contains the bits of several elements.

Bloom filters also allow light clients to quickly and privately obtain account information without downloading the entire blockchain.

Bloom filters are a powerful tool that allows additional innovation in blockchain applications and many other areas in the twenty first century.

11 Digital Signatures

Ethereum Classic (ETC) digital signatures secure transactions. These involve elliptic curve cryptography and the Elliptic Curve Digital Signature Algorithm (ECDSA). I will describe ETC digital signatures without these topics using only small Python functions.

Signing and verifying will be implemented using the following four constants and three functions:

```
N  = 115792089237316195423570985008687907852837564279074904382605163141518161494337
P  = 115792089237316195423570985008687907853269984665640564039457584007908834671663
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424

def invert(number, modulus):
        """
        Finds the inverses of natural numbers.
        """

        result = 1
        power  = number
        for e in bin(modulus - 2)[2:][::-1]:
                if int(e):
                        result = (result * power) % modulus
                power = (power ** 2) % modulus

        return result

def add(pair_1, pair_2):
        """
        Finds the sums of two pairs of natural numbers.
        """

        if   pair_1 == [0, 0]:
                result = pair_2
        elif pair_2 == [0, 0]:
                result = pair_1
        else:
```

(continues on next page)

(continued from previous page)

```
                if pair_1 == pair_2:
                        temp = 3 * pair_1[0] ** 2
                        temp = (temp * invert(2 * pair_1[1], P)) % P
                else:
                        temp = pair_2[1] - pair_1[1]
                        temp = (temp * invert(pair_2[0] - pair_1[0], P)) % P
                result = (temp ** 2 - pair_1[0]  - pair_2[0]) % P
                result = [result, (temp * (pair_1[0] - result) - pair_1[1]) % P]

        return result

def multiply(number, pair):
        """
        Finds the products of natural numbers and pairs of natural numbers.
        """

        result = [0, 0]
        power  = pair[:]
        for e in bin(number)[2:][::-1]:
                if int(e):
                        result = add(result, power)
                power = add(power, power)

        return result
```

The invert function defines an operation on numbers in terms of other numbers referred to as moduli. The add function defines an operation on pairs of numbers. The multiply function defines an operation on a number and a pair of numbers. Here are examples of their usage:

```
>>> invert(82856, 7164661)
3032150

>>> add([84672, 5768], [15684, 471346])
[98868508778765247164450388534339365517943901419260061027507991295919394382071,␣
↪1105310199765960047925915496510851918907114825918410403778324204643760261432231]

>>> multiply(82716, [31616, 837454])
[82708077205483544970470074583740846828577431856187364454411787387343982212318,␣
↪3083679665627566325654266299089016366217109228170420811810759116742388858830 4]
```

Private keys are any nonzero numbers less than the constant N. Public keys are the products of these private keys and the pair (Gx, Gy). For example:

```
>>> private_key = 296921718

>>> multiply(private_key, (Gx, Gy))
[29493341745186804828936410559976490896704930101972775917156948978213464516647,␣
↪1412058395951450305281641406861132868682763858156833529661587523540212231982 4]
```

Notice that public keys are pairs of numbers.

Signing transactions involves an operation on the Keccak 256 hashes of the transactions and private keys. The following function implements this operation:

```
import random

def sign(hash, priv_key):
        """
        Signs the hashes of transactions.
        """

        result = [0, 0]
        while (0 in result) or (result[1] > N / 2):
                temp      = random.randint(1, N - 1)
                result[0] = multiply(temp, (Gx, Gy))[0] % N
                result[1] = invert(temp, N) * (hash + priv_key * result[0])
                result[1] = result[1] % N

        return result
```

For example:

```
>>> hash = 0xf62d00f14db9521c03a39c20e94aa10a82ff5f5a614772b25e36757a95a71048

>>> private_key = 296921718

>>> sign(hash, private_key)
[126760036752790009956774124313990047605763110521262578877159318821644276868 66,␣
↪178539290279426111768393902157481575990529910880423567907461293386533424773 82]

>>> sign(hash, private_key)
[187833244646333877348260422959118029410260091088761307007271568962102033561 79,␣
↪419595629511572358943966601207711583320328041448675951961945814393454500085 33]
```

Notice that digital signatures are pairs of numbers. Notice also that the sign function can give different results for the same inputs!

Verifying digital signatures involves confirming certain properties with regards to the Keccak 256 hashes and public keys. The following function implements these checks:

```
def verify(sig, hash, pub_key):
        """
        Verifies the signatures of the hashes of transactions.
        """

        temp_1 = multiply((invert(sig[1], N) * hash)   % N, (Gx, Gy))
        temp_2 = multiply((invert(sig[1], N) * sig[0]) % N, pub_key)
        sum    = add(temp_1, temp_2)
        test_1 = (0 < sig[0] < N) and (0 < sig[1] < N)
        test_2 = sum != [0, 0]
        test_3 = sig[0] == sum[0] % N

        return test_1 and test_2 and test_3
```

For example:

```
>>> hash = 0xf62d00f14db9521c03a39c20e94aa10a82ff5f5a614772b25e36757a95a71048
```

(continues on next page)

(continued from previous page)

```
>>> private_key = 296921718

>>> public_key = multiply(private_key, (Gx, Gy))

>>> public_key
[29493341745186804828936410559976490896704930101972775917156948978213464516647,␣
↪14120583959514503052816414068611328686827638581568335296615875235402122319824]

>>> signature = sign(hash, private_key)

>>> signature
[54728868372105873293629977757277092827353030346967592768173610703187933361202,␣
↪18974025727476367931183775600389145833964496722266015570370178285290252701715]

>>> verify(signature, hash, public_key)
True
```

To verify that public keys correspond to specific ETC account addresses, confirm that the rightmost 20 bytes of the public key Keccak 256 hashes equal those addresses.

Strictly speaking, ETC digital signatures include additional small numbers referred to as recovery identifiers. These allow public keys to be determined solely from the signed transactions.

I have explained ETC digital signatures using code rather than mathematics. Hopefully seeing how signing and verifying can be implemented with these tiny functions has been useful.

12 How Nodes Find Each Other

Ethereum Classic (ETC) network nodes have accurate information about the network in spite of it being decentralized and constantly changing. I will describe how this happens.

Some network nodes are always available and accepting of new connections from other network nodes. These are referred to as bootstrap nodes. New network nodes first connect to bootstrap nodes to obtain information. Here is the current Geth bootstrap node list with network nodes specified using "enode" strings:

```
enode://
↪e809c4a2fec7daed400e5e28564e23693b23b2cc5a019b612505631bbe7b9ccf709c1796d2a3d29ef2b045f210caf51e3c4f5
↪112.32.157:30303
enode://
↪6e538e7c1280f0a31ff08b382db5302480f775480b8e68f8febca0ceff81e4b19153c6f8bf60313b93bef2cc34d34e1df4131
↪206.67.235:30303
enode://
↪5fbfb426fbb46f8b8c1bd3dd140f5b511da558cd37d60844b525909ab82e13a25ee722293c829e52cb65c2305b1637fa9a2ea
↪243.55.45:30303
enode://
↪42d8f29d1db5f4b2947cd5c3d76c6d0d3697e6b9b3430c3d41e46b4bb77655433aeedc25d4b4ea9d8214b6a43008ba6719937
↪155.176.151:30303
enode://
↪814920f1ec9510aa9ea1c8f79d8b6e6a462045f09caa2ae4055b0f34f7416fca6facd3dd45f1cf1673c0209e0503f02776b8f
↪154.136.117:30303
enode://
↪72e445f4e89c0f476d404bc40478b0df83a5b500d2d2e850e08eb1af0cd464ab86db6160d0fde64bd77d5f0d33507ae190356
↪198.71.200:30303
```

(continues on next page)

(continued from previous page)

```
enode://
↪5cd218959f8263bc3721d7789070806b0adff1a0ed3f95ec886fb469f9362c7507e3b32b256550b9a7964a23a938e8d42d45a
↪187.57.94:30303
enode://
↪39abab9d2a41f53298c0c9dc6bbca57b0840c3ba9dccf42aa27316addc1b7e56ade32a0a9f7f52d6c5db4fe74d8824bcedfea
↪76.238.49:30303
enode://
↪f50e675a34f471af2438b921914b5f06499c7438f3146f6b8936f1faeb50b8a91d0d0c24fb05a66f05865cd58c24da3e664d0
↪76.238.49:30306
enode://
↪6dd3ac8147fa82e46837ec8c3223d69ac24bcdbab04b036a3705c14f3a02e968f7f1adfcdb002aacec2db46e625c04bf8b5a1
↪237.131.102:30303
```

All enode strings contain Elliptic Curve Digital Signature Algorithm (ECDSA) public keys and internet sockets (addresses and ports).

Additional network nodes are found by network nodes storing and sharing lists of network nodes that have recently communicated with them. Ignoring older activity decreases the probability of sharing information about nonexistent network nodes. Network nodes continually communicate and thereby continually share their information. The whole process starts with bootstrap nodes and quickly includes many other network nodes.

Here is the current list for my Parity ETC node:

```
enode://
↪9fe33f0ebc5b0ce51879afa3f767b2a180536dafb34b5af24cef11bb1c136b90d7839d6340d912ccd1f8e917a9e24d0d908ca
↪215.240.147:30303
enode://
↪e1520f00ff23e82c87411964a70c08e77592aab16647ddc2b53a5617808330184aaaac786002d31e40a1db6ec1447d6b0d8eb
↪86.120.213:30303
enode://
↪a68a96163c842f1175d8eb515ca60ab93e80ab581cded1a11527f53e89f1f1cfb624e3f0a79a5dd76ee0fad54758ec9515be3
↪101.169.110:30303
enode://
↪e5ddf2ea2373697136681eaba314039ce60b99656c4eefabb2d01032d77dae384919941589f1d9309c340a854310b556d0595
↪125.218.87:30303
enode://
↪f5d269ebbfb94494e7e2251a49f430df5d7f510cf04173dad1229b12e4929e6a65f4c76cf9cb0f789c30f7a9d2e50a453a64b
↪230.160.215:30303
enode://
↪efce38b6ee1baa4fa0a48c4202cc175fe4668a376365bbd0b9735a06de04593c822e9064d6664e346af5c98efc0dd0e4f3f1b
↪98.232.156:30303
enode://
↪c38fadc7d03341aaa856c41d8af9733d535cbbde3e3e103dd97c1bf6a0e15f8a6ed77c7bfb04784d9be7bdef884172a1fca83
↪97.72.201:30303
enode://
↪10a0e3e2e4d9be6eab3615cd441da32dcebc7d51df9639c92eae35a7a434cf2c8e2bec756573ca9f49f48d6ed65917d4fbd22
↪125.156.101:30303
enode://
↪c1e0e9f8607afc20b70dc6f2b19258e879841561360385c63d004a9cdb1a93f1122e061ad405ff98a03f3413a945c9299e19d
↪57.166.35:30303
enode://
↪f4a1153780ccb0b4e2c86bdf11837035f621ddc09bcf7d874a9879bc20adf5a32ce0b5cda91674873ad4cd89ca0b6da6a8919
↪114.236.10:30303
```

(continues on next page)

(continued from previous page)

```
enode://
↪7c253d4172cb27a7c514c35ee1cad1ff0fa1a6d2d2ca1c3a6f67d7416173bf0f36c64d6caf5f3cf13700e81104b10642787c1
↪125.206.176:30303
enode://
↪ed81e7f825a5309c3c93698a440055f10fa617342f6e8c62645b3ee813515c488addee22a3175223b4d0bc410c0b3f0a2bc25
↪236.81.109:30307
enode://
↪8d45cb061b744f444d38c58f2fd972214f565e24539eb5ccd85612290a66d9da885a1012576784ab2a2cb93050a4ef99e6ba0
↪202.174.161:30303
enode://
↪1275a4224ee8ee7cedc5ab62a118428f6f4ec2841440cefacf8d01c368ac345d687052c667204e52cc22f0b85de0fc8195ace
↪22.91.166:30303
enode://
↪c60ae3bccd9ccded51ad5e8c2aa6bad8fad072c1809779566c85ca17f2f5d810fd6c11761e892c45787e9f27a213cc9ade324
↪100.182.189:40404
enode://
↪feaf167dad6e117bc07c507b8cf4aa1978cdcc592d218bde7782c67abc53dbf02ea8aa6799946aee4f8e5f73c58dbc28eadaa
↪125.25.106:30305
enode://
↪ba8efe932cbf32f0d8ee9a0af45ade59dd26e81cafdc86497e57aca0ae5e9bcc64ee058a4c844bc0d06ceea2db062f3777058
↪9.6.244:30303
enode://
↪c989f4ceea49188426561d2a1833c157cfd137948ae054ab47e770f4032b7708ac59121e13d0c267c23779b96cae258d39e4c
↪30.37.160:30305
enode://
↪4329eb2c7e62206421469d413ddc6f8bdf10eb176aaec065282bd31d198ace82871034fa4af34ad0ed5051d2d9d82845a6729
↪114.236.167:30303
enode://
↪485f2c2368cfb6614026058e37c4d1a63aba7285df77663198c96824dc9b44b2d3861198d42d8c00f7f2d3ee7a00619ebb6e2
↪221.229.254:30303
enode://
↪78f960f4cc378980409704957818e55bb671775d3ca3f731fb8a0468a1156b7158a42e921f4077eaf9de3e8a6905360fca629
↪65.3.132:30303
enode://
↪5eb58e0bda31307b19b6d25350bba7b812fc0337cb34e79a5796bbdcaf3bace0535d56e2f485cab7839d03754d5bacb98d666
↪91.28.47:30305
enode://
↪ad132e9609f8288c07e8958af5b1c77dad6001af774e5a5b3d3e06bd339b8a52e63a70ba92df917ddaa1494ed228c735cc5be
↪131.14.202:30303
enode://
↪f926fb79061578191b8d125d6fca889f711c94d1dab19f3c106671812a098bb7837b5b40c67f8f0ae2bd0a0410cfa57a00270
↪78.23.204:30303
enode://
↪a38b8841524f4a0f6e4161511ef9ed60b7da1a5303a316fd99d997c5f2642313eab3cbe560b1c62dab1ac9be8e92fe61611c4
↪227.151.104:30303
```

Amazingly, ETC nodes can find each other within a headless and ever changing system. There are many such brilliant riches to appreciate in the ETC design when one looks.

13 Code Is Law Principle

The code is law principle is the principle that no one has the right to censor the execution of code on the ETC blockchain.

14 A Crypto-Decentralist Manifesto By Bit Novosti

Blockchains are going to rule the world, providing a mechanism for scaling social and economic cooperation to an unprecedented level—a truly global scale. Such cooperation will involve not only human beings, groups and associations but also a growing multitude of increasingly independent artificial agents.

Every blockchain creates a social network around its applications, with network value growing exponentially with the number of participants in accordance with Reed's Law. This value isn't extracted by intermediaries or controllers, as with previous centralized models. Instead, it's shared among participants, providing economic incentives for cooperation without coercion.

Not all blockchains are created equal. There are three key characteristics that make scalable blockchain-enabled cooperation possible: openness, neutrality and immutability.

Openness is necessary. It goes without saying that the rules of the game should be open for anyone to see and understand. Anyone should be able to participate in any layer of the system without asking for any permission whatsoever. Anyone should be able to use the network in accordance with its rules. Anyone should be able to create their own client implementing the open protocol. Anyone should be able to contribute to network security, and so on. No registration, identification or other preconditions should limit participation. All such limitations prevent the network from scaling and their enforcement creates centralization risks.

Neutrality is necessary. It's important for anyone participating in blockchain-enabled cooperation to be on an equal footing with everyone else. It doesn't matter if you wield huge economic power or only a tiny amount. It doesn't matter whether you're a saintly Mother Theresa or a vicious drug dealer. It doesn't matter whether you're a human or a refrigerator. It doesn't matter what you believe in, what political theory you subscribe to, or whether you're a moral or immoral person. A participant's ethnicity, age, sex, profession, social standing, friends or affiliations, make or model, goals, purposes or intentions—none of this matters to the blockchain even a bit. The rules of the game are exactly the same for everyone, period. Without neutrality, the system is skewed towards one set of participants at the expense of others. In that case, it's less likely to gain universal acceptance and maximize network value for everyone.

Immutability is necessary. The blockchain is a truth machine preserving one universally accepted version of history, one immutable sequence of events. What's true once is always true, regardless of political or business interests, and no amount of lobbying can change that. If it's simply not possible to change history, then no resources are wasted on the effort. If there are any loopholes at all, then sufficiently motivated and determined interest groups will exploit them at the expense of others, diminishing network value for everyone.

The rules governing the blockchain network are known in advance. They're exactly the same for everyone and not subject to change other than with 100% consensus. Yes, it must be 100%. Because any change to the system's rules that not all participants freely agree to creates a network split, diminishing network value for everyone.

It's impossible to achieve these blockchain characteristics without the system **being truly decentralized.** If any aspect of the blockchain system becomes subject to centralized control, this introduces an attack vector enabling the violation of one or more of the key blockchain characteristics. It may be possible to limit participation (such as by enforcing AML/KYC rules), thus violating openness. It may be possible to enforce discriminatory policies (such as by filtering "illegal" transactions), thus violating neutrality. It may be possible to rewrite the history of events (such as by confiscating or "redistributing" funds), thus violating

immutability. Introducing centralized chokepoints creates a precondition for the introduction of "blockchain intermediaries or controllers" who can siphon value out of the system at other participants' expense.

So decentralization is the most important feature of blockchain systems, the one everything else depends on. With decentralization, blockchains will come to rule the world. Without it, they'll be "contained" and railroaded into niche applications.

We decentralists are committed to keeping blockchains open, neutral and immutable. We're committed to keeping blockchain systems decentralized. This informs all our actions and positions towards any developments in the crypto world and beyond. All attempts to violate any of the key blockchain characteristics should be fought. All changes to a blockchain's rules that introduce new centralization risks or strengthen existing ones should be fought. Only developments that are clearly beneficial to decentralization or strengthen the three key blockchain characteristics should be supported and encouraged.

The blockchain revolution won't be centralized. Let's make sure of it.

Onward.

15 The Ethereum Classic Declaration Of Independence

Let it be known to the entire world that on July 20th, 2016, at block 1,920,000, we as a community of sovereign individuals stood united by a common vision to continue the original Ethereum blockchain that is truly free from censorship, fraud or third party interference. In realizing that the blockchain represents absolute truth, we stand by it, supporting its immutability and its future. We do not make this declaration lightly, nor without forethought to the consequences of our actions.

Looking Back

It should be stated with great gratitude that we acknowledge the creation of the Ethereum blockchain platform by the Ethereum Foundation and its founding developers. It certainly can be said without objection, that without their hard work and dedication that we as a community would not be where we are today.

From its inception, the Ethereum blockchain was presented as a decentralized platform for "applications that run exactly as programmed without any chance of fraud, censorship, or third-party interference" [1] . It provided a place for the free association of ideas and applications from across the globe without fear of discrimination while also providing pseudonymity. In this decentralized platform, many of us saw great promise.

List of Grievances

It is however, with deep regret, that we as a community have had to spontaneously organize [2] to defend the Ethereum blockchain platform from its founding members and organization due to a long train of abuses, specifically by the leadership of the Ethereum Foundation. These grievances are as follows.

- For rushing the creation of a "soft fork," which was comprised of a minor change in the Ethereum blockchain code for the sole purpose of creating a blacklist and censoring transactions that normally would have been allowed.
- For neglecting the full implications of the "soft fork" by the Ethereum blockchain as a warning that they were violating the principles and values coded therein. [3]
- For creating an unrepresentative voting mechanism called the "carbon vote", which they initially stated was "unofficial" [4] only to contradict these statements a day before determining to hard fork. [5]
- For rushing the creation of a "hard fork," which was comprised of an irregular state change in the Ethereum blockchain code that violated the properties of immutability, fungibility, and the sanctity of the ledger.

- For willfully deciding to not include replay protection in the "hard fork", an action which has unnecessarily cost exchanges and thousands of users the rightful ownership of their Ether tokens. [6]

Respecting the Values Essential for Blockchains

One might ask what harm can be done from changing the code of the Ethereum blockchain and bailing out [7] "The DAO" token holders, which is not an unreasonable question. Many of us have an innate sense of right and wrong, so at first glance rescuing "The DAO" felt right. However, it violated two key aspects of what gives peer-to-peer cash [8] and smart contract-based systems value: fungibility and immutability.

Immutability means the blockchain is inviolable. That only valid transactions agreed upon via a cryptographic protocol determined by mathematics are accepted by the network. Without this, the validity of all transactions could come into question, since if the blockchain is mutable, any transaction could be modified. Not only does this leave transactions open to fraud, but it might spell disaster for any distributed application running atop the platform.

Fungibility is the feature of money where one unit equals another unit. For instance, a Euro equals another Euro just as a Bitcoin equals another Bitcoin. Unfortunately, an ETH no longer equals another ETH. The alleged attacker's ETH was no longer as good as your ETH and was worthy of censorship, deemed necessary by a so-called majority.

Ultimately, these breaches in fungibility and immutability were made possible by the subjective morality judgements of those who felt a burning desire to bring the alleged attacker to justice. However, in doing so they compromised a core pillar of Ethereum just to do what they felt was in the interests of the "greater good". In a global community where each individual has their own laws, customs, and beliefs, who is to say what is right and wrong?

Deeply alarmed that these core tenets were disregarded by many of the Foundation's developers, and a sizable portion of Ethereum participants, we, as a community, have organized and formed a code of principles to follow for the Ethereum Classic chain.

The Ethereum Classic Code of Principles

We believe in a decentralized, censorship-resistant, permission-less blockchain. We believe in the original vision of Ethereum as a world computer that cannot be shut down, running irreversible smart contracts. We believe in a strong separation of concerns, where system forks of the codebase are only possible when fixing protocol level vulnerabilities, bugs, or providing functionality upgrades. We believe in the original intent of building and maintaining a censorship-resistant, trustless and immutable development platform.

Herein are written the declared values by which participants within the Ethereum Classic community agree. We encourage that these principles not be changed via edict by any individual or faction claiming to wield power, authority or credibility to do so.

We, as a community agree that:

- The purpose of Ethereum Classic is to provide a decentralized platform that runs decentralized applications which execute exactly as programmed without any possibility of downtime, censorship, fraud or third party interference.
- Code is law; there shall be no changes to the Ethereum Classic code that violate the properties of immutability, fungibility, or sanctity of the ledger; transactions or ledger history cannot for any reason be reversed or modified.
- Forks and/or changes to the underlying protocol shall only be permitted for updating or upgrading the technology on which Ethereum Classic operates.
- Internal project development can be funded by anyone, whether via a trusted third party of their choice or directly, using the currency of their choice on a per project basis and following a transparent, open and decentralized crowdfunding protocol.

- Any individual or group of individuals may propose improvements, enhancements, or upgrades to existing or proposed Ethereum Classic assets.
- Any individual or group of individuals may use the Ethereum Classic decentralized platform to build decentralized applications, hold crowdsales, create autonomous organisations/corporations, or for any other purpose they deem suitable.

Looking Forward

For the many reasons listed above, we have chosen to rename the original blockchain "Ethereum Classic" with the ticker symbol "ETC", so that the community and all other participants can identify the original, unforked, and immutable blockchain platform.

Our most sincere gratitude goes to those developers within and outside the Foundation who opposed interfering with the Ethereum blockchain ledger and enabled the Ethereum Classic chain to survive and live on. We know there are many of you and we welcome you at anytime to join our decentralized community.

We will continue the vision of decentralized governance for the Ethereum Classic blockchain and maintain our opposition to any centralized leadership takeover, especially by the Ethereum Foundation as well as the developers who have repeatedly stated that they would no longer develop the Ethereum Classic chain.

We likewise will openly resist the "tyranny of the majority," and will not allow the values of the system to be compromised. As a united community, we will continue to organize for the defense and advancement, as required, for the continuation and assurance of this grand experiment. The Ethereum Classic platform, its code and technology, are now open to the world as Open Source software. [9] It is now freely available for all who wish to improve and build upon it: a truly free and trustless world computer that we together as a community have proven and will continue to prove is anti-fragile. [10]

- The Ethereum Classic Community

16 Glossary

51% attack attacks against blockchain systems that are possible if attackers control over half of the mining resources

account data structure associated with Ethereum Classic users and smart contracts

address unique numbers that identify blockchain accounts and are derived from the associated private keys

ASIC application specific integrated circuit, devices optimized for specific tasks such as blockchain mining

block sets of blockchain transactions and related logistical information

block explorer website presenting information about a blockchain

block header all the fields of Ethereum Classic blocks except the transaction and uncle header lists

block propagation distribution of copies of blocks throughout blockchain networks

blockchain linear arrays of blocks each of which met the requirements of the blockchain system

bootstrap nodes Ethereum Classic network computers that are always available and accepting of new connections from other network computers

classic ether the native cryptocurrency of Ethereum Classic, also referred to as ether

coinbase blockchain account that receives mining rewards

compiler programs that translate programs from one programming language to others

consensus agreements among blockchain miners regarding the selection of the official chain

cryptocurrency money systems implemented on blockchain systems

daemon autonomous software agents

DAG directed acyclic graph, used in the Ethereum Classic proof of work calculations

dapp applications that run on blockchain systems, also referred to as decentralized applications

decentralization the elimination of centralized control entities from software and network designs

decentralized application applications that run on blockchain systems, also referred to as dapps

decentralized autonomous enterprise sophisticated smart contracts that perform many of the functions of organizations

difficulty difficulty of a proof of work calculation in a blockchain system

digital signature strings associated with other strings that prove the creators has access to a private key

digital signature authentication mechanism in which text snippets appended to data establishing that the creator had access to the associated private key and did a calculation on the data with it,

elliptic curve cryptography type of cryptography typically used in blockchain public and private key systems

encryption a cryptographic process used to protect the privacy of information

ether the native cryptocurrency of Ethereum Classic, also referred to as classic ether

event requests in smart contracts to log some information

EVM Ethereum Virtual Machine

exchange place to buy and sell cryptocurrencies

external account one of the two types of Ethereum Classic accounts, used by external clients

fast syncing updating blockchain copies and downloading instead of calculating state information

gas units used to measure resource usage in the Ethereum Classic system

gas limit limits on the maximum possible of gas units, blocks for example have limit on the amount of gas all their transactions can require

gas price prices of gas units in classic ether

genesis block first blocks of blockchains

GHOST Greedy Heaviest Observed Subtree blockchain protocol

hash fixed length string calculated from another possibly much longer string with many uses such as confirming data integrity and naming objects

hashrate total mining capacity of a blockchain as measured by how fast hashes can be calculated

hexadecimal compact method of representing numbers involving numbers and letters, is especially used for big numbers

IPC interprocess communication, used to communicate between processes on a computer

keyfile file containing a private and public key pair, typically encrypted

light client blockchain client that submit that can utilize a blockchain but does not maintain a complete copy of the blochchain

Merkle Patricia trie data structures which represents key value pairs

message smart contract account requests to other smart contract accounts, may transfer funds as well as invoke or create smart contracts

mining process of creating, verifying and distributing blocks in a blockchain

mining pool group of miners working together

mining reward new cryptocurrency tokens given to miners for performing mining

node computers in a network such as a blockchain network

node discovery process of finding other nodes in a network such as blockchain network

nonce numbers used once (Number ONCE) for various purposes in blockchain systems such as proof of work calculations and preventing replay attacks

peer to peer network decentralized network without a centralized control entity

private key secret numbers of blockchain accounts used to authorize transactions from it

proof of work results of difficult calculations in some blockchain systems used to increase security

protocol set of rules to accomplish something like a process in a blockchain system

public key pairs of numbers derived from the private keys of blockchain accounts used to identify blockchain accounts and determine account addresses

receipt transaction logs

RPC remote procedure call, used to invoke code on the same or different computers

serialization process of converting a data structure into a linear array of bits

Serpent high level smart contract language which is similar to Python

sharding mechanism to increase the scalability of blockchain systems by dividing blockchains into subsets (shards) managed by different parts of the network

sidechain blockchain that is associated with but still separate from another blockchain

smart contract autonomous software agents such as software running on blockchains

Solidity high level smart contract language which is similar to Javascript

state in Ethereum Classic, all account information for all accounts at some point in time

syncing updating blockchain copies

transaction external account requests to the Ethereum Classic system, may transfer funds as well as invoke or create smart contracts

transaction fee gas costs to execute transactions

trustless property of public blockchain system whereby no entity has special permissions

uncle losing blocks in mining contests that are used to increase the security of blockchain systems

virtual machine computing resource implemented in software

Vyper high level smart contract language which is similar to Python

wallet set of public and private keys, may also refer to other associated information and software

web 3 name that refers to the vision of a more secure, trustless blockchain based World Wide Web replacement