# ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

**BERLIN VERSION 630ed56 – 2022-02-13**

DR. GAVIN WOOD FOUNDER, ETHEREUM & PARITY

[GAVIN@PARITY.IO](mailto:GAVIN@PARITY.IO)

Abstract. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. Introduction

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bit- coin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographi- cally secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original “currency application” of the technology into other applications albeit rather simplistic ones.

Ethereum is a project which attempts to build the gen- eralised technology; technology on which all transaction- based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto un- explored compute paradigm in the mainstream: a trustful object messaging compute framework.

* 1. **Driving Factors.** There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompati- bility, incompetence, unwillingness, expense, uncertainty, inconvenience, or corruption of existing legal systems. By specifying a state-change system through a rich and unam- biguous language, and furthermore architecting a system such that we can reasonably expect that an agreement will be thus enforced autonomously, we can provide a means to this end.

Dealings in this proposed system would have several attributes not often found in the real world. The incorrupt- ibility of judgement, often difficult to find, comes naturally from a disinterested algorithmic interpreter. Transparency, or being able to see exactly how a state or judgement came about through the transaction log and rules or instructional codes, never happens perfectly in human-based systems since natural language is necessarily vague, information

is often lacking, and plain old prejudices are difficult to shake.

Overall, we wish to provide a system such that users can be guaranteed that no matter with which other indi- viduals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

* 1. **Previous Work.** [Buterin [2013a]](#_bookmark73) first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a block- chain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains un- changed.

[Dwork and Naor [1992]](#_bookmark91) provided the first work into the usage of a cryptographic proof of computational expendi- ture (“proof-of-work”) as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a *strong economic signal*, allowing a receiver to make a physical assertion without having to rely upon *trust*. [Back](#_bookmark69) [[2002]](#_bookmark69) later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by [Vish-](#_bookmark107) [numurthy et al.](#_bookmark107) [[2003].](#_bookmark107) In this instance, the token was used to keep peer-to-peer file trading in check, providing “consumers” with the ability to make micro-payments to “suppliers” for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record couldn’t be corrupted and that malicious actors could not spoof payment or unjustly complain about service deliv- ery. Five years later, [Nakamoto [2008]](#_bookmark87) introduced another such proof-of-work-secured value token, somewhat wider in scope. The fruits of this project, Bitcoin, became the first widely adopted global decentralised transaction ledger.

Other projects built on Bitcoin’s success; the alt-coins introduced numerous other currencies through alteration to the protocol. Some of the best known are Litecoin and Primecoin, discussed by [Sprankel [2013].](#_bookmark101) Other projects sought to take the core value content mechanism of the pro- tocol and repurpose it; [Aron [2012]](#_bookmark70) discusses, for example,

1

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 2

the Namecoin project which aims to provide a decentralised name-resolution system.

Other projects still aim to build upon the Bitcoin net- work itself, leveraging the large amount of value placed in the system and the vast amount of computation that goes into the consensus mechanism. The Mastercoin project, first proposed by [Willett](#_bookmark109) [[2013],](#_bookmark109) aims to build a richer protocol involving many additional high-level features on top of the Bitcoin protocol through utilisation of a number of auxiliary parts to the core protocol. The Coloured Coins

itself—that would be far too big). They also punctuate the transaction series with incentives for nodes to *mine*. This incentivisation takes place as a state-transition function, adding value to a nominated account.

Mining is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. This scheme is known as a proof-of-work and is discussed in detail in section [11.5.](#_bookmark68)

Formally, we expand to:

project, proposed by [Rosenfeld et al. [2012],](#_bookmark89) takes a similar but more simplified strategy, embellishing the rules of a transaction in order to break the fungibility of Bitcoin’s base currency and allow the creation and tracking of tokens through a special “chroma-wallet”-protocol-aware piece of

(2)

(3)

(4)

***σ****t*+1 ≡ [Π(](#_bookmark67)***σ****t, B*)

*B* ≡ (*...,* (*T*0*, T*1*, ...*)*, ...*)

Π(***σ****, B*) ≡ [Ω(](#_bookmark65)*B,* [Υ(Υ(](#_bookmark0)***σ****, T*0)*, T*1)*...*)

Where [Ω](#_bookmark65) is the block-finalisation state transition func-

software.

Additional work has been done in the area with discard- ing the decentralisation foundation; Ripple, discussed by [Boutellier and Heinzen [2014],](#_bookmark74) has sought to create a “fed- erated” system for currency exchange, effectively creating a new financial clearing system. It has demonstrated that high efficiency gains can be made if the decentralisation premise is discarded.

Early work on smart contracts has been done by [Szabo](#_bookmark103) [[1997]](#_bookmark103) and [Miller [1997].](#_bookmark86) Around the 1990s it became clear that algorithmic enforcement of agreements could become a significant force in human cooperation. Though no specific system was proposed to implement such a system, it was proposed that the future of law would be heavily affected by such systems. In this light, Ethereum may be seen as a general implementation of such a *crypto-law* system.

For a list of terms used in this paper, refer to Appen- dix [A.](#_bookmark111)

1. The Blockchain Paradigm

Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a gen- esis state and incrementally execute transactions to morph it into some current state. It is this current state which we accept as the canonical “version” of the world of Ethereum. The state can include such information as account bal- ances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admis- sible. Transactions thus represent a valid arc between two states; the ‘valid’ part is important—there exist far more invalid state changes than valid state changes. Invalid state changes might, e.g., be things such as reducing an account balance without an equal and opposite increase elsewhere. A valid state transition is one which comes about through a transaction. Formally:

tion (a function that rewards a nominated party); [*B*](#_bookmark19) is this block, which includes a series of transactions amongst some other components; and [Π](#_bookmark67) is the block-level state-transition function.

This is the basis of the blockchain paradigm, a model that forms the backbone of not only Ethereum, but all decentralised consensus-based transaction systems to date.

* 1. **Value.** In order to incentivise computation within the network, there needs to be an agreed method for transmit- ting value. To address this issue, Ethereum has an intrinsic currency, Ether, known also as ETH and sometimes referred to by the Old English ¯D. The smallest subdenomination of Ether, and thus the one in which all integer values of the currency are counted, is the Wei. One Ether is defined as being 1018 Wei. There exist other subdenominations of Ether:

Multiplier Name

100 Wei

1012 Szabo

1015 Finney

1018 Ether

Throughout the present work, any reference to value, in the context of Ether, currency, a balance or a payment, should be assumed to be counted in Wei.

* 1. **Which History?** Since the system is decentralised and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks. In order to form a consensus as to which path, from root [(the genesis bloc](#_bookmark153)k) to leaf (the block containing the most recent transactions) through this tree structure, known as the blockchain, there must be an agreed-upon scheme. If there is ever a disagreement between nodes as to which root-to-leaf path down the block tree is the ‘best’ blockchain, then a *fork* occurs.

(1) ***σ***

*t*+1

≡ Υ(***σ****t, T* )

This would mean that past a given point in time (block), multiple states of the system may coexist: some nodes be-

where Υ is the Ethereum state transition function. In Ethereum, Υ, together with ***σ*** are considerably more pow- erful than any existing comparable system; Υ allows com- ponents to carry out arbitrary computation, while ***σ*** allows components to store arbitrary state between transactions. Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of refer- ence. Blocks function as a journal, recording a series of transactions together with the previous block and an iden- tifier for the final state (though do not store the final state

lieving one block to contain the canonical transactions, other nodes believing some other block to be canonical, potentially containing radically different or incompatible transactions. This is to be avoided at all costs as the un- certainty that would ensue would likely kill all confidence in the entire system.

The scheme we use in order to generate consensus is a simplified version of the GHOST protocol introduced by [Sompolinsky and Zohar [2013].](#_bookmark100) This process is described in detail in section [10.](#_bookmark61)

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 3

Sometimes, a path follows a new protocol from a par- ticular height (block number). This document describes one version of the protocol, namely the *Berlin* version defined by [Beiko et al. [2021b].](#_bookmark72) In order to follow back the history of a path, one must reference multiple versions of this document. Here are the block numbers of protocol updates on the Ethereum main network:

|  |  |
| --- | --- |
| Name | First Block Number |
| *F*Homestead | 1150000 |
| *F*TangerineWhistle | 2463000 |
| *F*SpuriousDragon | 2675000 |
| *F*Byzantium | 4370000 |
| *F*Constantinople | 7280000 |
| *F*Petersburg | 7280000 |
| *F*Istanbul | 9069000 |
| *F*MuirGlacier | 9200000 |
| *F*Berlin | 12244000 |
| *F*London | 12965000 |
| *F*ArrowGlacier | 13773000 |

Occasionally actors do not agree on a protocol change, and a permanent fork occurs. In order to distinguish be- tween diverged blockchains, EIP-155 by [Buterin](#_bookmark79) [[2016b]](#_bookmark79) introduced the concept of chain ID, which we denote by *β*.

For the Ethereum main network

(5) *β* = 1

1. Conventions

We use a number of typographical conventions for the formal notation, some of which are quite particular to the present work:

The two sets of highly structured, ‘top-level’, state val- ues, are denoted with bold lowercase Greek letters. They fall into those of world-state, which are denoted ***σ*** (or a variant thereupon) and those of machine-state, ***µ***.

Functions operating on highly structured values are denoted with an upper-case Greek letter, e.g. [Υ,](#_bookmark0) the Ethereum state transition function.

For most functions, an uppercase letter is used, e.g. *C*, the general cost function. These may be subscripted to denote specialised variants, e.g. [*C*SSTORE](#_bookmark146), the cost func- tion for the [SSTORE](#_bookmark145) operation. For specialised and possibly

externally defined functions, we may format as typewriter text, e.g. the Keccak-256 hash function (as per version 3 of the winning entry to the SHA-3 contest by [Bertoni](#_bookmark75) [et al. [2011],](#_bookmark75) rather than the final SHA-3 specification), is

denoted KEC (and generally referred to as plain Keccak). Also, KEC512 refers to the Keccak-512 hash function.

Tuples are typically denoted with an upper-case letter,

e.g. *T* , is used to denote an Ethereum transaction. This symbol may, if accordingly defined, be subscripted to refer to an individual component, e.g. [*T*n](#_bookmark40), denotes the nonce of said transaction. The form of the subscript is used to

denote its type; e.g. uppercase subscripts refer to tuples with subscriptable components.

Scalars and fixed-size byte sequences (or, synonymously, arrays) are denoted with a normal lower-case letter, e.g. *n* is used in the document to denote a [transaction nonce.](#_bookmark40) Those with a particularly special meaning may be Greek,

e.g. *δ*, the number of items required on the stack for a given operation.

Arbitrary-length sequences are typically denoted as a bold lower-case letter, e.g. **o** is used to denote the byte sequence given as the output data of a message call. For particularly important values, a bold uppercase letter may be used.

Throughout, we assume scalars are non-negative inte- gers and thus belong to the set N. The set of all byte sequences is B, formally defined in Appendix [B.](#_bookmark112) If such a set of sequences is restricted to those of a particular length, it is denoted with a subscript, thus the set of all

byte sequences of length 32 is named B32 and the set of all non-negative integers smaller than 2256 is named N256. This is formally defined in section [4.3.](#_bookmark19)

Square brackets are used to index into and reference individual components or subsequences of sequences, e.g. ***µ*s**[0] denotes the first item on the machine’s stack. For subsequences, ellipses are used to specify the intended range, to include elements at both limits, e.g. ***µ*m**[0*..*31] denotes the first 32 items of the machine’s memory.

In the case of the global state ***σ***, which is a sequence of accounts, themselves tuples, the square brackets are used to reference an individual account.

When considering variants of existing values, we follow the rule that within a given scope for definition, if we assume that the unmodified ‘input’ value be denoted by the placeholder then the modified and utilisable value is

denoted as , and intermediate values would be ∗, ∗∗

&c. On very particular occasions, in order to maximise readability and only if unambiguous in meaning, we may use alpha-numeric subscripts to denote intermediate values, especially those of particular note.

When considering the use of existing functions, given a function *f* , the function [*f* ∗](#_bookmark27) denotes a similar, element-wise version of the function mapping instead between sequences. It is formally defined in section [4.3.](#_bookmark19)

We define a number of useful functions throughout. One of the more common is *.€*, which evaluates to the last item in the given sequence:

(6) *.€*(**x**) ≡ **x**[ll**x**ll − 1]

1. Blocks, State and Transactions

Having introduced the basic concepts behind Ethereum, we will discuss the meaning of a transaction, a block and the state in more detail.

* 1. **World State.** The world state (*state*), is a map- ping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix [B).](#_bookmark112) Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modi- fied Merkle Patricia tree (*trie*, see Appendix [D).](#_bookmark117) The trie requires a simple database backend that maintains a map- ping of byte arrays to byte arrays; we name this underlying database the state database. This has a number of benefits; firstly the root node of this structure is cryptographically dependent on all internal data and as such its hash can be used as a secure identity for the entire system state. Secondly, being an immutable data structure, it allows any previous state (whose root hash is known) to be recalled by simply altering the root hash accordingly. Since we store all such root hashes in the blockchain, we are able to trivially revert to old states.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 4

The account state, ***σ***[*a*], comprises the following four fields:

**nonce:** A scalar value equal to the number of trans- actions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For ac- count of address *a* in state ***σ***, this would be for- mally denoted ***σ***[*a*]n.

**balance:** A scalar value equal to the number of Wei

owned by this address. Formally denoted ***σ***[*a*]b.

**storageRoot:** A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage con- tents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted ***σ***[*a*]s.

**codeHash:** The hash of the EVM code of this

account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corre- sponding hashes for later retrieval. This hash is formally denoted ***σ***[*a*]c, and thus the code may be

denoted as **b**, given that KEC(**b**) = ***σ***[*a*]c.

Since we typically wish to refer not to the trie’s root hash but to the underlying set of key/value pairs stored within, we define a convenient equivalence:

1. TRIE *L*∗I (***σ***[*a*]**s**) ≡ ***σ***[*a*]s

The collapse function for the set of key/value pairs in the trie, *L*∗I , is defined as the element-wise transformation of the base function *L*I, given as:

1. *L*I (*k, v*) ≡ KEC(*k*)*,* RLP(*v*) where:

(9) *k* ∈ B32 ∧ *v* ∈ N

It shall be understood that ***σ***[*a*]**s** is not a ‘physical’ member of the account and does not contribute to its later serialisation.

If the **codeHash** field is the Keccak-256 hash of the empty string, i.e. ***σ***[*a*]c = KEC () , then the node represents a simple account, sometimes referred to as a “non-contract” account.

Thus we may define a world-state collapse function *L*S:

(10) *L*S(***σ***) ≡ {*p*(*a*) : ***σ***[*a*] ∅}

where

(11) *p*(*a*) ≡ KEC(*a*)*,* RLP (***σ***[*a*]n*,* ***σ***[*a*]b*,* ***σ***[*a*]s*,* ***σ***[*a*]c)

This function, *L*S, is used alongside the trie function to provide a short identity (hash) of the world state. We assume:

(12) ∀*a* : ***σ***[*a*] = ∅ ∨ (*a* ∈ B20 ∧ *v*(***σ***[*a*])) where *v* is the account validity function:

(13) *v*(*x*) ≡ *x*n ∈ N256∧*x*b ∈ N256∧*x*s ∈ B32∧*x*c ∈ B32

An account is *empty* when it has no code, zero nonce and zero balance:

(14)

EMPTY(***σ****, a*) ≡ ***σ***[*a*]c = KEC () ∧***σ***[*a*]n = 0∧***σ***[*a*]b = 0

Even callable precompiled contracts can have an empty account state. This is because their account states do not usually contain the code describing its behavior.

An account is *dead* when its account state is non-existent or empty:

(15) DEAD(***σ****, a*) ≡ ***σ***[*a*] = ∅ ∨ EMPTY(***σ****, a*)

* 1. **The Transaction.** A transaction (formally, *T* ) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. The sender of a transaction cannot be a contract. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemi- nation[1](#_bookmark10). EIP-2718 by [Zoltu [2020]](#_bookmark110) introduced the notion of different transaction types. As of the Berlin version of the protocol, there are two transaction types: 0 (legacy) and 1 (EIP-2930 by [Buterin and Swende [2020b]).](#_bookmark80) Further, there are two subtypes of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as ‘contract creation’). All transaction types specify a number of common fields:

**type:** EIP-2718 transaction type; formally *T*x.

**nonce:** A scalar value equal to the number of trans- actions sent by the sender; formally *T*n.

**gasPrice:** A scalar value equal to the number of

Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of this transaction; formally *T*p.

**gasLimit:** A scalar value equal to the maximum

amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally *T*g.

**to:** The 160-bit address of the message call’s recipi-

ent or, for a contract creation transaction, ∅, used here to denote the only member of B0 ; formally *T*t.

**value:** A scalar value equal to the number of Wei to

be transferred to the message call’s recipient or, in the case of contract creation, as an endowment to the newly created account; formally *T*v.

**r, s:** Values corresponding to the signature of the

transaction and used to determine the sender of the transaction; formally *T*r and *T*s. This is ex- panded in Appendix [F.](#_bookmark126)

EIP-2930 (type 1) transactions also have:

**accessList:** List of access entries to warm up; for- mally *T***A**. Each access list entry *E* is a tuple of an account address and a list of storage keys:

*E* ≡ (*E*a*, E***s**).

**chainId:** Chain ID; formally *T*c. Must be equal to the network chain ID [*β*](#_bookmark1).

**yParity:** Signature Y parity; formally *T*y.

1Notably, such ‘tools’ could ultimately become so causally removed from their human-based initiation—or humans may become so causally-neutral—that there could be a point at which they rightly be considered autonomous agents. e.g. contracts may offer bounties to humans for being sent transactions to initiate their execution.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 5

Legacy transactions do not have an **accessList** (*T***A** = ()), while **chainId** and **yParity** for legacy transactions are combined into a single value:

**w:** A scalar value encoding Y parity and possi- bly chain ID; formally *T*w. *T*w = 27 + *T*y or *T*w = 2*β*+35+*T*y (see EIP-155 by [Buterin [2016b]).](#_bookmark79)

Additionally, a contract creation transaction (regardless whether legacy or EIP-2930) contains:

**init:** An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally *T***i**.

**init** is an EVM-code fragment; it returns the **body**, a second fragment of code that executes each time the account receives a message call (either through a trans- action or due to the internal execution of code). **init** is executed only once at account creation and gets discarded immediately thereafter.

In contrast, a message call transaction contains:

**data:** An unlimited size byte array specifying the input data of the message call, formally *T***d**.

Appendix [F](#_bookmark126) specifies the function, *S*, which maps trans- actions to the sender, and happens through the ECDSA of the SECP-256k1 curve, using the hash of the transaction (excepting the latter three signature fields) as the datum to sign. For the present we simply assert that the sender of a given transaction *T* can be represented with *S*(*T* ).

(16)

f

(*T*n*, T*p*, T*g*, T*t*, T*v*,* **p***, T*w*, T*r*, T*s) if *T*x = 0

*L*T(*T* ) ≡

(*T , T , T , T , T , T ,* **p***, T*

*, T , T , T* ) if *T*

= 1

**parentHash:** The Keccak 256-bit hash of the parent block’s header, in its entirety; formally *H*p.

**ommersHash:** The Keccak 256-bit hash of the om-

mers list portion of this block; formally *H*o.

**beneficiary:** The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally *H*c.

**stateRoot:** The Keccak 256-bit hash of the root

node of the state trie, after all transactions are executed and finalisations applied; formally *H*r.

**transactionsRoot:** The Keccak 256-bit hash of the

root node of the trie structure populated with each transaction in the transactions list portion of the block; formally *H*t.

**receiptsRoot:** The Keccak 256-bit hash of the root

node of the trie structure populated with the re- ceipts of each transaction in the transactions list portion of the block; formally *H*e.

**logsBloom:** The Bloom filter composed from index-

able information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally *H*b.

**difficulty:** A scalar value corresponding to the dif-

ficulty level of this block. This can be calculated from the previous block’s difficulty level and the timestamp; formally *H*d.

**number:** A scalar value equal to the number of an-

cestor blocks. The genesis block has a number of zero; formally [*H*i](#_bookmark30).

**gasLimit:** A scalar value equal to the current limit

where

c n p g t v

**A** y r s x

of gas expenditure per block; formally *H*l.

**gasUsed:** A scalar value equal to the total gas used

(17) **p** *T***i** if *T*t = ∅

f≡

*T***d** otherwise

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the access list *T***A** and the arbitrary length byte arrays *T***i** and *T***d**.

(18) *T*x ∈ {0*,* 1} ∧ *T*c = *β* ∧ *T*n ∈ N256 ∧ *T*p ∈ N256 ∧ *T*g ∈ N256 ∧ *T*v ∈ N256 ∧ *T*w ∈ N256 ∧ *T*r ∈ N256 ∧ *T*s ∈ N256 ∧ *T*y ∈ N1 ∧ *T***d** ∈ B ∧ *T***i** ∈ B

where

(19) Nn = {*P* : *P* ∈ N ∧ *P <* 2*n*}

The address hash *T***t** is slightly different: it is either a 20-byte address hash or, in the case of being a contract- creation transaction (and thus formally equal to ∅), it is

the RLP empty byte sequence and thus the member of B0:

in transactions in this block; formally *H*g.

**timestamp:** A scalar value equal to the reasonable output of Unix’s time() at this block’s inception; formally [*H*s](#_bookmark32).

**extraData:** An arbitrary byte array containing data

relevant to this block. This must be 32 bytes or fewer; formally *H*x.

**mixHash:** A 256-bit hash which, combined with the

nonce, proves that a sufficient amount of compu- tation has been carried out on this block; formally *H*m.

**nonce:** A 64-bit value which, combined with the mix-

hash, proves that a sufficient amount of computa- tion has been carried out on this block; formally [*H*n](#_bookmark33).

1. *T*

∈ fB20 if *T*t = ∅

The other two components in the block are simply a list

B0 otherwise

**t**

* 1. **The Block.** The block in Ethereum is the collec- tion of relevant pieces of information (known as the block *header* ), *H*, together with information corresponding to the comprised transactions, **T**, and a set of other block headers **U** that are known to have a parent equal to the present block’s parent’s parent (such blocks are known as *ommers*[2](#_bookmark20)). The block header contains several pieces of information:

of ommer block headers (of the same format as above),

*B***U** and a series of the transactions, *B***T**. Formally, we can refer to a block *B*:

1. *B* ≡ (*B*H*, B***T***, B***U**)

2*ommer* is a gender-neutral term to mean “sibling of parent”; see [https://nonbinary.miraheze.org/wiki/Gender\_neutral\_language\_in\_](https://nonbinary.miraheze.org/wiki/Gender_neutral_language_in_English%23Aunt/Uncle) [English#Aunt/Uncle](https://nonbinary.miraheze.org/wiki/Gender_neutral_language_in_English%23Aunt/Uncle)

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 6

* + 1. *Transaction Receipt.* In order to encode information about a transaction concerning which it may be useful to form a zero-knowledge proof, or index and search, we encode a receipt of each transaction containing certain in- formation from its execution. Each receipt, denoted *B***R**[*i*] for the *i*th transaction, is placed in an index-keyed [trie](#_bookmark118) and the root recorded in the header as [*H*e](#_bookmark24).

The transaction receipt, *R*, is a tuple of five items com-

prising: the type of the transaction, *R*x, the status code of the transaction, *R*z, the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened, *R*u, the set of logs created through execution of the transaction, [*R***l**](#_bookmark116)and the Bloom filter composed from information in those logs, [*R*b](#_bookmark114):

(22) *R* ≡ (*R*x*, R*z*, R*u*, R*b*, R***l**)

* + 1. *Holistic Validity.* We can assert a block’s validity if and only if it satisfies several conditions: it must be in- ternally consistent with the ommer and transaction block hashes and the given transactions *B***T** (as specified in sec [11),](#_bookmark63) when executed in order on the base state ***σ*** (derived from the final state of the parent block), result in a new state of the identity *H*r:

(33) *H*r ≡ TRIE(*LS* (Π(***σ****, B*))) ∧ *H*o ≡ KEC(RLP(*L*∗*H* (*B***U**))) ∧ *H*t ≡ TRIE({∀*i <* ll*B***T**ll*, i* ∈ N :

*p*T(*i, B***T**[*i*])}) ∧

*H*e ≡ TRIE({∀*i <* ll*B***R**ll*, i* ∈ N :

*p*R(*i, B***R**[*i*])}) ∧

*H*b ≡

**r**∈*B***R**

**r**b

*R*x is equal to the [type](#_bookmark4) of the corresponding transaction.

The function *L*R prepares a transaction receipt for being transformed into an RLP-serialised byte array:

1. *L*R(*R*) ≡ (*R*z*, R*u*, R*b*, R***l**)

We assert that the status code *R*z is a non-negative

where *p*T(*k, v*) and *p*R(*k, v*) are pairwise RLP transforma- tions, but with a special treatment for EIP-2718 transac- tions:

(34)

f ≡

*p* (*k, T* ) RLP(*k*)*,* RLP([*L*T](#_bookmark12)(*T* )) if *T*x = 0

T

(*T*x) · RLP(*L*T(*T* )) otherwise

integer:

1. *R*z ∈ N

We assert that *R*u, the cumulative gas used, is a non- negative integer and that the logs Bloom, *R*b, is a hash of

and

(35)

*p*R(*k, R*) ≡

RLP(*k*)*,*

RLP([*L*R](#_bookmark26)(*R*)) if *R*x = 0

(*R*x) · RLP(*L*R(*R*)) otherwise

f

size 2048 bits (256 bytes):

1. *R*u ∈ N ∧ *R*b ∈ B256

The sequence *R***l** is a series of log entries, (*O*0*, O*1*, ...*). A log entry, *O*, is a tuple of the logger’s address, *Oa*, a possibly empty series of 32-byte log topics, *O***t** and some number of bytes of data, *O***d**:

(26) *O* ≡ (*O*a*,* (*O***t**0*, O***t**1*, ...*)*, O***d**)

(27) *O*a ∈ B20 ∧ ∀*x* ∈ *O***t** : *x* ∈ B32 ∧ *O***d** ∈ B

We define the Bloom filter function, *M* , to reduce a log entry into a single 256-byte hash:

(· is the concatenation of byte arrays).

Furthermore:

(36) TRIE(*L*S(***σ***)) = *P* (*BH* )*H* r

Thus TRIE(*L*S(***σ***)) is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of the state ***σ*** with values encoded using RLP, and *P* (*B*H) is the parent block of *B*, defined directly.

The values stemming from the computation of transac- tions, specifically the [transaction receipts,](#_bookmark21) *B***R**, and that defined through the transaction’s [state-accumulation func-](#_bookmark67) [tion, Π,](#_bookmark67) are formalised later in section [11.4.](#_bookmark64)

* + 1. *Serialisation.* The function *L*B and *L*H are the prepa-

(28) *M* (*O*) ≡

*x*∈{*O*a}∪*O***t**

*M*3:2048(*x*)

ration functions for a block and block header respectively. We assert the types and order of the structure for when

where *M*3:2048 is a specialised Bloom filter that sets three

the RLP transformation is required:

bits out of 2048, given an arbitrary byte sequence. It does

this through taking the low-order 11 bits of each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence.[3](#_bookmark28) Formally:

(37)

(38)

*L*H(*H*) ≡ ( *H*p*, H*o*, H*c*, H*r*, H*t*, H*e*, H*b*, H*d*,*

*H*i*, H*l*, H*g*, H*s*, H*x*, H*m*, H*n )

*L*B(*B*) ≡ *L*H(*B*H)*, L* ∗T(*B***T**)*, L*∗H([*B***U**](#_bookmark18))

(29)*M*

f

|  |  |  |
| --- | --- | --- |
| 3:2048 B) | | ≡ |
| (30) | **y** | = |
| (31) | ∀*i* ∈ {0*,* 2*,* 4} | : |
| (32) | *m*(**x***, i*) | ≡ |

(**x** : **x** ∈

**y** : **y** ∈ B256 where: (0*,* 0*, ...,* 0) except:

where *L* T takes a special care of EIP-2718 transactions:

B2047−*m*(**x***,i*)(**y**) = 1

KEC(**x**)[*i, i* + 1] mod 2048

(39)

*L* T

(*T* ) = [*L*T](#_bookmark12)(*T* ) if *T*x = 0

(*T*x) · RLP(*L*T(*T* )) otherwise

where B is the bit reference function such that Bj(**x**) equals the bit of index *j* (indexed from 0) in the byte array **x**. Notably, it treats **x** as big-endian (more significant bits

will have smaller indices).

with *L*∗T and *L*∗H being element-wise sequence transforma- tions, thus:

(40)

*f* ∗ (*x*0*, x*1*, ...*) ≡ *f* (*x*0)*, f* (*x*1)*, ...* for any function *f*

311 bits = 22048, and the low-order 11 bits is the modulo 2048 of the operand, which is in this case is “each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence.”

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 7

The component types are defined thus:

(41) [*H*p](#_bookmark13) ∈ B32 ∧ *H*o ∈ B32 ∧ *H*c ∈ B20 ∧ [*H*r](#_bookmark23) ∈ B32 ∧ *H*t ∈ B32 ∧ [*H*e](#_bookmark24) ∈ B32 ∧ [*H*b](#_bookmark25) ∈ B256 ∧ *H*d ∈ N ∧ [*H*i](#_bookmark30) ∈ N ∧ [*H*l](#_bookmark29) ∈ N ∧ *H*g ∈ N ∧ [*H*s](#_bookmark32) ∈ N256 ∧ [*H*x](#_bookmark15) ∈ B ∧ *H*m ∈ B32 ∧ [*H*n](#_bookmark17) ∈ B8

where

(42) Bn = {*B* : *B* ∈ B ∧ ll*B*ll = *n*}

We now have a rigorous specification for the construc- tion of a formal block structure. The RLP function RLP (see Appendix [B)](#_bookmark112) provides the canonical method for trans- forming this structure into a sequence of bytes ready for transmission over the wire or storage locally.

[[2016a].](#_bookmark77) In the *Byzantium* release, with EIP-649, the ice age was delayed by creating a fake block number, *H*i, which is obtained by subtracting three million from the actual block number, which in other words reduced *E* and the time difference between blocks, in order to allow more time to develop proof-of-stake and preventing the network from “freezing” up. Subsequently, EIP-1234 by [Schoedon](#_bookmark94) [[2018],](#_bookmark94) EIP-2384 by [Conner [2019],](#_bookmark83) EIP-3554 by [Hancock](#_bookmark102) [[2021],](#_bookmark102) and EIP-4345 by [Beiko et al. [2021a]](#_bookmark71) increased the subtrahend *κ*.

The canonical gas limit *H*l of a block of header *H* must fulfil the relation:

* + 1. *Block Header Validity.* We define *P* (*B*H parent block of *B*, formally:

(43) *P* (*H*) ≡ *B*

: KEC(RLP(*B*H )) = [*H*p](#_bookmark13)

) to be the

(53)

*H*l *< P* (*H*)Hl

*H*

l

*> P* (*H*)

Hl

+ *P* (*H*)Hl

1024

∧

*P* (*H*)

−

Hl

1024

∧

The block number is the parent’s block number incre- mented by one:

(44) *H*i ≡ *P* (*H*)Hi + 1

The canonical difficulty of a block of header *H* is defined as *D*(*H*):

(45)

f234 if *H*i = 0

*H*l ;;: 5000

*H*s is the timestamp (in Unix’s time()) of block *H* and must fulfil the relation:

*D*(*H*) ≡

where:

max *D*

min

*, P* (*H*)Hd

+ *x* × *ς*2

+ *E* otherwise

(54) *H*s *> P* (*H*)Hs

(46)

(47)

(48)

*D*min ≡ 217

*x P* (*H*)Hd 2048

≡

*ς*2 ≡ max (*y* −

f≡

*H*s − *P* (*H*)Hs

9

*,* −99

This mechanism enforces a homeostasis in terms of the time between blocks; a smaller period between the last two blocks results in an increase in the difficulty level and thus additional computation required, lengthening the likely next period. Conversely, if the period is too large, the

(49)

(50)

(51)

*y* 1 if ll*P* (*H*)**U**ll = 0

2 otherwise

*E* ≡ 2l*H*i*t* ÷100000J−2

*H*i ≡ max(*H*i − *κ,* 0)

|  |  |  |  |
| --- | --- | --- | --- |
| (52) |  |  | (55) *n H*  d |
| *κ* ≡ | ****3000000 | if  if if if | *F*Byzantium *H*i *< F*Constantinople  *F*Constantinople *H*i *< F*MuirGlacier  *F*MuirGlacier *H*i *< F*London with (*n, m*) = PoW(*H*n*, H F*London *H*i *< F*ArrowGlacier Where *H*n is the new bl |

difficulty, and expected time to the next block, is reduced.

The nonce, [*H*n](#_bookmark17), must satisfy the relations:

2256

5000000

****

9000000

9700000

****

10700000 if *H*i

;;: *F*

ArrowGlacier

∧ *m* = *H*m

n*,* **d**).

ock’s header *H*, but *without* the nonce and mix-hash components, **d** being the current DAG, a large data set needed to compute the mix-hash, and PoW

The *Homestead* difficulty parameter, *ς*2, is used to affect

a dynamic homeostasis of time between blocks, as the time between blocks varies, as discussed below, as implemented in EIP-2 by [Buterin](#_bookmark76) [[2015].](#_bookmark76) In the Homestead release, the exponential difficulty symbol, *E* causes the difficulty to slowly increase (every 100,000 blocks) at an exponential rate, and thus increasing the block time difference, and putting time pressure on transitioning to proof-of-stake. This effect, known as the “difficulty bomb”, or “ice age”, was explained in EIP-649 by [Schoedon and Buterin [2017]](#_bookmark95)

and delayed and implemented earlier in EIP-2. *ς*2 was also modified in EIP-100 with the use of *x*, the adjustment factor above, and the denominator 9, in order to target the mean block time including uncle blocks by [Buterin](#_bookmark77)

is the proof-of-work function (see section [11.5):](#_bookmark68) this evalu-

ates to an array with the first item being the mix-hash, to prove that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on *H* and **d**. Given an approximately uniform distribution in the range [0*,* 264), the expected time to find a solution is proportional to the difficulty, *H*d.

This is the foundation of the security of the blockchain and is the fundamental reason why a malicious node can- not propagate newly created blocks that would otherwise overwrite (“rewrite”) history. Because the nonce must sat- isfy this requirement, and because its satisfaction depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult and,

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 8

over time, requires approximately the total compute power of the trustworthy portion of the mining peers.

Thus we are able to define the block header validity function *V* (*H*):

256

2

1. *V* (*H*) ≡ *n H* ∧ *m* = *H*m ∧

d

*H*d = *D*(*H*) ∧

*H*g ≤ *H*l ∧

∧

* 1. Transaction Execution

The execution of a transaction is the most complex part of the Ethereum protocol: it defines the state transition function [Υ.](#_bookmark0) It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include:

* 1. The transaction is well-formed RLP, with no addi- tional trailing bytes;
  2. the transaction signature is valid;

*H*l *< P* (*H*)Hl

+ *P* (*H*)Hl

1024

*P* (*H*)Hl

*H*l *> P* (*H*)Hl −

1024

* 1. the [transaction nonce](#_bookmark40) is valid (equivalent to the [sender account’s current nonce);](#_bookmark3)
  2. the sender account has no contract code deployed

∧

(see EIP-3607 by [Feist et al. [2021]);](#_bookmark93)

*H*l ;;: 5000 ∧

*H*s *> P* (*H*)Hs ∧

*H*i = *P* (*H*)Hi + 1 ∧

ll*H*xll ≤ 32

where (*n, m*) = PoW(*H*n*, H*n*,* **d**)

Noting additionally that **extraData** must be at most 32 bytes.

5. Gas and Payment

In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing complete- ness, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of *gas* (see Ap- pendix [G](#_bookmark131) for the fees associated with various computation). Thus any given fragment of programmable computation (this includes creating contracts, making message calls, utilising and accessing account storage and executing op- erations on the virtual machine) has a universally agreed cost in terms of gas.

Every transaction has a specific amount of gas associ- ated with it: **gasLimit**. This is the amount of gas which is implicitly purchased from the sender’s account balance. The purchase happens at the according **gasPrice**, also specified in the transaction. The transaction is consid- ered invalid if the account balance cannot support such a purchase. It is named **gasLimit** since any unused gas at the end of the transaction is refunded (at the same rate of purchase) to the sender’s account. Gas does not exist outside of the execution of a transaction. Thus for accounts with trusted code associated, a relatively high gas limit may be set and left alone.

In general, Ether used to purchase gas that is not re- funded is delivered to the *beneficiary* address, the address of an account typically under the control of the miner. Transactors are free to specify any **gasPrice** that they wish, however miners are free to ignore transactions as they choose. A higher gas price on a transaction will there- fore cost the sender more in terms of Ether and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners. Miners, in general, will choose to advertise the minimum gas price for which they will execute transactions and transactors will be free to canvas these prices in determining what gas price to offer. Since there will be a (weighted) distribution of min- imum acceptable gas prices, transactors will necessarily have a trade-off to make between lowering the gas price and maximising the chance that their transaction will be mined in a timely manner.

* 1. the gas limit is no smaller than the intrinsic gas,

*g*0, used by the transaction; and

* 1. the sender account balance contains at least the cost, *v*0, required in up-front payment.

Formally, we consider the function [Υ,](#_bookmark0) with *T* being a transaction and ***σ*** the state:

(57) ***σ*** = Υ(***σ****, T* )

Thus ***σ*** is the post-transactional state. We also define [Υg](#_bookmark42) to evaluate to the amount of gas used in the execution of a transaction, [Υ**l**](#_bookmark42) to evaluate to the transaction’s accrued log items and [Υz](#_bookmark42) to evaluate to the status code resulting from the transaction. These will be formally defined later.

* 1. **Substate.** Throughout transaction execution, we ac- crue certain information that is acted upon immediately following the transaction. We call this the *accrued transac- tion substate*, or *accrued substate* for short, and represent it as *A*, which is a tuple:

(58) *A* ≡ (*A***s***, A***l***, A***t***, A*r*, A***a***, A***K**)

The tuple contents include *A***s**, the self-destruct set: a set of accounts that will be discarded following the trans- action’s completion. *A***l** is the log series: this is a series of archived and indexable ‘checkpoints’ in VM code execution that allow for contract-calls to be easily tracked by onlook- ers external to the Ethereum world (such as decentralised application front-ends). *A***t** is the set of touched accounts, of which the empty ones are deleted at the end of a transac- tion. *A*r is the refund balance, increased through using the

[SSTORE](#_bookmark145) instruction in order to reset contract storage to

zero from some non-zero value. Though not immediately refunded, it is allowed to partially offset the total execution costs. Finally, EIP-2929 by [Buterin and Swende [2020a]](#_bookmark81) introduced *A***a**, the set of accessed account addresses, and *A***K**, the set of accessed storage keys (more accurately, each element of *A***K** is a tuple of a 20-byte account address and a 32-byte storage slot).

We define the empty accrued substate *A*0 to have no self-destructs, no logs, no touched accounts, zero refund bal-

ance, all precompiled contracts in the accessed addresses, and no accessed storage:

(59) *A*0 ≡ (∅*,* ()*,* ∅*,* 0*, π,* ∅)

where [*π*](#_bookmark51)is the set of all precompiled addresses.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 9

* 1. **Execution.** We define intrinsic gas *g*0, the amount of gas this transaction requires to be paid prior to execution, as follows:

accrued substate *A* and status code *z*: (66)

****[Λ](#_bookmark44)4(***σ***0*, A*∗*, S*(*T* )*, S*(*T* )*, g,*

(***σ***

P

*, g, A, z*) ≡ 

*T*p*, T*v*, T***i***,* 0*,* ∅*,* T) if *T*t = ∅

∗

*g*0 ≡

(60)

f[*G*txdatazero](#_bookmark136) if *i* = 0

[Θ](#_bookmark47)4(***σ***0*, A , S*(*T* )*, S*(*T* )*, T*t*,*

 *T*t*, g, T*p*, T*v*, T*v*, T***d***,* 0*,* T) otherwise

****

*i*∈*T***i***,T***d**

f

[*G*txdatanonzero](#_bookmark137) otherwise

where

+ [*G*txcreate](#_bookmark135) if *T*t = ∅

0 otherwise

(67)

(68)

*A*∗ ≡ *A*0 except

*A*∗**a** ≡ *A*0 ∪ {*S*(*T* )} ∪*E*∈*T*

{[*E*](#_bookmark9)a}

+ [*G*transaction](#_bookmark138)

*T***A** −1

(69)

**a**

*A*∗**K** ≡ L

**A**

{∀*i <* ll*E***s**ll*, i* ∈ N : (*E*a*, E***s**[*i*])}

+

*j*=0

*G*accesslistaddress + ll*T***A**[*j*]**s**ll*G*accessliststorage

*E*∈*T***A**

and *g* is the amount of gas remaining after deducting

where *T***i***, T***d** means the series of bytes of the transaction’s associated data and initialisation EVM-code, depending on whether the transaction is for contract-creation or message-

call. *G*txcreate is added if the transaction is contract- creating, but not if a result of EVM-code. [*G*accesslistaddress](#_bookmark132) and [*G*accessliststorage](#_bookmark133) are the costs of warming up account and storage access, respectively. *G* is fully defined in Ap-

pendix [G.](#_bookmark131)

The up-front cost *v*0 is calculated as:

(61) *v*0 ≡ [*T*g](#_bookmark6)[*T*p](#_bookmark5) + [*T*v](#_bookmark7)

The validity is determined as:

the basic amount required to pay for the existence of the transaction:

1. *g* ≡ *T*g − *g*0

Note we use [Θ](#_bookmark47)4 and [Λ](#_bookmark44)4 to denote the fact that only the first four components of the functions’ values are taken; the final represents the message-call’s output value (a byte array) and is unused in the context of transaction evalua-

tion.

After the message call or contract creation is processed, the refund counter has to be incremented for the accounts that were self-destructed throughout its invocation.

1. *A*r ≡ *A*r + *R*selfdestruct

*i*∈*A***s**

(62) *S*(*T* ) = ∅

***σ***[*S*(*T* )]c = KEC ()

∧

Then the state is finalised by determining the amount

*T*n = ***σ***[*S*(*T* )]n ∧

∧

*g*0 *T*g ∧

*v*0 ***σ***[*S*(*T* )]b ∧

to be refunded, *g*∗ from the remaining gas, *g*, plus some

allowance from the refund counter, to the sender at the original rate.

*T*g *B*Hl − [*.€*](#_bookmark2)(*B***R**)u

Note the final condition; the sum of the transaction’s gas

1. *g*∗ ≡ *g*

+ min

*T*g − *g*

2

*, A*r

limit, *T*g, and the gas utilised in this block prior, given by

[*.€*](#_bookmark2)(*B***R**)u, must be no greater than the block’s **gasLimit**, *B*Hl. Also, with a slight abuse of notation, we assume that ***σ***[*S*(*T* )]c = KEC () , ***σ***[*S*(*T* )]n = 0, and ***σ***[*S*(*T* )]b = 0 if ***σ***[*S*(*T* )] = ∅.

The execution of a valid transaction begins with an irrevocable change made to the state: the [nonce of the](#_bookmark3) [account of the sender,](#_bookmark3) *S*(*T* ), is incremented by one and the balance is reduced by part of the up-front cost, *T*g*T*p. The

The total refundable amount is the legitimately remain-

ing gas *g*, added to [*A*r](#_bookmark37), with the latter component being capped up to a maximum of half (rounded down) of the total amount used *T*g − *g*. Therefore, *g*∗ is the total gas that remains after the transaction has been executed.

The Ether for the gas is given to the miner, whose address is specified as the beneficiary of the present block

*B*. So we define the pre-final state ***σ***∗ in terms of the

provisional state ***σ***P:

gas available for the proceeding computation, *g*, is defined

as *T*g − *g*0. The computation, whether contract creation

***σ***∗ ≡ ***σ***P except

or a message call, results in an eventual state (which may legally be equivalent to the current state), the change to which is deterministic and never invalid: there can be no invalid transactions from this point.

***σ***∗[*S*(*T* )]b ≡ ***σ***P[*S*(*T* )]b + *g*∗*T*p

***σ***∗[*m*]b ≡ ***σ***P[*m*]b + (*T*g − *g*∗)*T*p *m* ≡ *B*Hc

We define the checkpoint state ***σ***0:

The final state, ***σ***, is reached after deleting all accounts

that either appear in the self-destruct set or are touched

(63)

(64)

***σ***0 ≡ ***σ*** except:

***σ***0[*S*(*T* )]b ≡ ***σ***[*S*(*T* )]b − *T*g*T*p

and empty:

***σ*** ≡ ***σ***∗ except

(65)

***σ***0[*S*(*T* )]n ≡ ***σ***[*S*(*T* )]n + 1

∀*i* ∈ *A***s** : ***σ*** [*i*] = ∅

Evaluating ***σ***P

from ***σ***0

depends on the transaction type;

∀*i* ∈ *A***t** : ***σ*** [*i*] = ∅ if DEAD(***σ***∗*, i*)

either contract creation or message call; we define the tuple of post-execution provisional state ***σ***P, remaining gas *g*,

And finally, we specify Υg, the total gas used in this transaction Υ**l**, the logs created by this transaction and

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 10

f

Υz, the status code of this transaction:

(90)

***σ***∗[*a*] = 1*, v* + *v,* [TRIE](#_bookmark118)(∅)*,* KEC ()

Υg(***σ****, T* ) ≡ *T*g − *g*∗

Υ**l**(***σ****, T* ) ≡ [*A***l**](#_bookmark38)

(91)

***σ***∗[*s*] =

∅ if ***σ***[*s*] = ∅ ∧ *v* = 0

**a**∗ otherwise

Υz(***σ****, T* ) ≡ *z*

(92)

**a**∗ ≡ (***σ***[*s*]n*,* ***σ***[*s*]b − *v,* ***σ***[*s*]**s***,* ***σ***[*s*]c)

These are used to help define the [transaction receipt](#_bookmark21) and are also used [later](#_bookmark66) for state and nonce validation.

* 1. Contract Creation

There are a number of intrinsic parameters used when creating an account: sender (*s*), original transactor[4](#_bookmark46) (*o*), available gas (*g*), gas price (*p*), endowment (*v*) together with an arbitrary length byte array, **i**, the initialisation EVM code, the present depth of the message-call/contract- creation stack (*e*), the salt for new account’s address (*ζ*) and finally the permission to make modifications to the state (*w*). The salt *ζ* might be missing (*ζ* = ∅); formally,

(83) *ζ* ∈ B32 ∪ B0

where *v* is the account’s pre-existing value, in the event it was previously in existence:

(93) *v* 0 if ***σ***[*a*] = ∅

f≡

***σ***[*a*]b otherwise

Finally, the account is initialised through the execution of the initialising EVM code **i** according to the execution model (see section [9).](#_bookmark52) Code execution can effect several events that are not internal to the execution state: the account’s storage can be altered, further accounts can be created and further message calls can be made. As such, the code execution function [Ξ](#_bookmark57) evaluates to a tuple of the resultant state ***σ***∗∗, available gas remaining *g*∗∗, the re- sultant accrued substate *A*∗∗ and the body code of the account **o**.

If the creation was caused by [CREATE2](#_bookmark150), then *ζ* ∅.

We define the creation function formally as the function Λ, which evaluates from these values, together with the state ***σ*** and the accrued substate *A*, to the tuple containing the new state, remaining gas, new accrued substate, status code and output (***σ****, g, A, z,* **o**):

(84) (***σ****, g, A, z,* **o**) ≡ Λ(***σ****, A, s, o, g, p, v,* **i***, e, ζ, w*)

The address of the new account is defined as being the rightmost 160 bits of the Keccak-256 hash of the [RLP](#_bookmark113) encoding of the structure containing only the sender and the [account nonce.](#_bookmark3) For [CREATE2](#_bookmark150) the rule is different and is described in EIP-1014 by [Buterin](#_bookmark82) [[2018].](#_bookmark82) Combining the two cases, we define the resultant address for the new account *a*:

(94) (***σ***∗∗*, g*∗∗*, A*∗∗*,* **o**) ≡ Ξ(***σ***∗*, g, A*∗*, I*)

where *I* contains the parameters of the [execution environ-](#_bookmark53) [men](#_bookmark53)t, that is:

(85) *a* ≡ ADDR(*s,* ***σ***[*s*]n − 1*, ζ,* **i**)

|  |  |  |  |
| --- | --- | --- | --- |
| (95) | *I*a | ≡ | *a* |
| (96) | *I*o | ≡ | *o* |
| (97) | *I*p | ≡ | *p* |
| (98) | *I***d** | ≡ | () |
| (99) | *I*s | ≡ | *s* |
| (100) | [*I*v](#_bookmark56) | ≡ | *v* |
| (101) | *I***b** | ≡ | **i** |
| (102) | *I*e | ≡ | *e* |
| (103) | *I*w | ≡ | *w* |

(86) ADDR(*s, n, ζ,* **i**) ≡ B96*..*255 KEC *L*A(*s, n, ζ,* **i**)

f

*I***d** evaluates to the empty tuple as there is no input data to this call. *I*H has no special treatment and is determined

(87)

*L*A(*s, n, ζ,* **i**) ≡

RLP (*s, n*) if *ζ* = ∅

(255) · *s* · *ζ* · KEC(**i**) otherwise

from the blockchain.

Code execution depletes gas, and gas may not go below zero, thus execution may exit before the code has come

where · is the concatenation of byte arrays, B*a..b*(*X*) evalu- ates to a binary value containing the bits of indices in the range [*a, b*] of the binary data *X*, and ***σ***[*x*] is the address state of *x*, or ∅ if none exists. Note we use one fewer than

the sender’s nonce value; we assert that we have incre- mented the sender account’s nonce prior to this call, and so the value used is the sender’s nonce at the beginning of the responsible transaction or VM operation.

The address of the new account is added to the set of accessed accounts:

1. *A*∗ ≡ *A* except *A*∗**a** ≡ *A***a** ∪ {*a*}

The account’s nonce is initially defined as one, the bal- ance as the value passed, the storage as empty and the code hash as the Keccak 256-bit hash of the empty string; the sender’s balance is also reduced by the value passed. Thus the mutated state becomes ***σ***∗:

1. ***σ***∗ ≡ ***σ*** except:

to a natural halting state. In this (and several other) ex- ceptional cases we say an out-of-gas (OOG) exception has occurred: The evaluated state is defined as being the empty set, ∅, and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation.

If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, *c*, proportional to the size of the created contract’s code:

(104) *c* ≡ *G*codedeposit × ll**o**ll

If there is not enough gas remaining to pay this, i.e.

*g*∗∗ *< c*, then we also declare an out-of-gas exception.

The gas remaining will be zero in any such exceptional condition, i.e. if the creation was conducted as the recep- tion of a transaction, then this doesn’t affect payment of the intrinsic cost of contract creation; it is paid regardless. However, the value of the transaction is not transferred to the aborted contract’s address when we are out-of-gas.

4which can differ from the sender in the case of a message call or contract creation not directly triggered by a transaction but coming from the execution of EVM-code

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 11

If such an exception does not occur, then the remaining gas is refunded to the originator and the now-altered state is allowed to persist. Thus formally, we may specify the resultant state, gas, accrued substate and status code as (***σ****, g, A, z*) where:

(105)

*g* 0 if *F*

f≡

*g*∗∗ − *c* otherwise

(106)

****

***σ*** if *F* ∨ ***σ***∗∗ = ∅

*******σ*** except:

∗∗

recipient (*r*), the account whose code is to be executed (*c*, usually the same as recipient), available gas (*g*), value (*v*) and gas price (*p*) together with an arbitrary length byte array, **d**, the input data of the call, the present depth of the message-call/contract-creation stack (*e*) and finally the permission to make modifications to the state (*w*).

Aside from evaluating to a new state and accrued transaction substate, message calls also have an extra component—the output data denoted by the byte array **o**. This is ignored when executing transactions, however mes- sage calls can be initiated due to VM-code execution and in this case this information is used.

(110)

(***σ****, g**, A**, z,* **o**) ≡ Θ(***σ****, A, s, o, r, c, g, p, v, v*˜*,* **d***, e, w*)

***σ*** ≡

(107)

***σ***[*a*] = ∅ if DEAD(***σ***∗∗*, a*)

***σ***∗∗ except:

****

 ***σ***[*a*]c = KEC(**o**) otherwise

Note that we need to differentiate between the value that

is to be transferred, *v*, from the value apparent in the execution context, *v*˜, for the DELEGATECALL instruction. We define ***σ***1, the first transitional state as the orig- inal state but with the value transferred from sender to

*A A*∗ if *F* ∨ ***σ***∗∗ = ∅

f

≡

*A*∗∗ otherwise

(108)

f≡

*z* 0 if *F* ∨ ***σ***∗∗ = ∅

1 otherwise

where (109)

*F* ≡ ***σ***[*a*] = ∅ ∧ ***σ***[*a*]*c* = KEC () ∨ ***σ***[*a*]*n* = 0 ∨

(***σ***∗∗ = ∅ ∧ **o** = ∅) ∨

*g*∗∗ *< c* ∨

recipient:

(111) ***σ***1[*r*]b ≡ ***σ***[*r*]b + *v* ∧ ***σ***1[*s*]b ≡ ***σ***[*s*]b − *v*

unless *s* = *r*.

Throughout the present work, it is assumed that if ***σ***1[*r*] was originally undefined, it will be created as an account with no code or state and zero balance and nonce. Thus the previous equation should be taken to mean:

(112) ***σ***1 ≡ ***σ***1 except:

ll**o**ll *>* 24576

(113) ***σ***1[*s*] ≡ f∅

if ***σ***1[*s*] = ∅

∧ *v* = 0

The exception in the determination of ***σ*** dictates that **o**, the resultant byte sequence from the execution of the initialisation code, specifies the final body code for the newly-created account.

Note that intention is that the result is either a suc- cessfully created new contract with its endowment, or no new contract with no transfer of value. In addition, ob-

serve that if the execution of the initialising code [reverts](#_bookmark151) (***σ***∗∗ = ∅ ∧ **o** = ∅), the resultant gas *g* is not depleted (provided there was no other exception), but no new ac-

count is created.

7.1. **Subtleties.** Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code[5](#_bookmark49). Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SELFDESTRUCT instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

* 1. Message Call

In the case of executing a message call, several param- eters are required: sender (*s*), transaction originator (*o*),

**a**1 otherwise

(114) **a**1 ≡ ***σ***1[*s*]n*,* ***σ***1[*s*]b − *v,* ***σ***1[*s*]**s***,* ***σ***1[*s*]c

(115) and ***σ***1 ≡ ***σ*** except:

(116)



***σ***1[*r*] ≡ (0*, v,* TRIE(∅)*,* KEC(())) if ***σ***[*r*] = ∅ ∧ *v*  = 0

***σ***1[*r*] ≡ ∅ if ***σ***[*r*] = ∅ ∧ *v* = 0



***σ***1[*r*] ≡ **a**1 otherwise

(117) **a**1 ≡ (***σ***[*r*]n*,* ***σ***[*r*]b + *v,* ***σ***[*r*]**s***,* ***σ***[*r*]c)

The account’s associated code (identified as the frag- ment whose Keccak-256 hash is ***σ***[*c*]c) is executed according to the execution model (see section [9).](#_bookmark52) Just as with con- tract creation, if the execution halts in an exceptional fashion (i.e. due to an exhausted gas supply, stack under- flow, invalid jump destination or invalid instruction), then no gas is refunded to the caller and the state is reverted to the point immediately prior to balance transfer (i.e. ***σ***).

5During initialization code execution, EXTCODESIZE on the address should return zero, which is the length of the code of the account while

CODESIZE should return the length of the initialization code (as defined in [H.2).](#_bookmark142)

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 12

where

1. Ξ ≡

and

***σ σ*** if ***σ***∗∗ = ∅

***σ***∗∗ otherwise



f

|  |  |  |
| --- | --- | --- |
| (118) |  | ≡ |
| (119) | *g* | ≡ |
| (120) |  | ≡ |
| (121) | *z* | ≡ |

0 if ***σ***∗∗ = ∅ ∧

**o** = ∅

f

*g*∗∗ otherwise

*A A* if ***σ***∗∗ = ∅

*A*∗∗ otherwise

f0 if ***σ***∗∗ = ∅

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| (122) (***σ***∗∗*, g*∗∗*, A*∗∗*,* | **o**) | ≡ | Ξ | 1 | otherwise |
| (123) | *I*a | ≡ | *r* |  |  |
| (124) | *I*o | ≡ | *o* |  |  |
| (125) | *I*p | ≡ | *p* |  |  |
| (126) | *I***d** | ≡ | **d** |  |  |
| (127) | *I*s | ≡ | *s* |  |  |
| (128) | *I*v | ≡ | *v*˜ |  |  |
| (129) | *I*e | ≡ | *e* |  |  |
| (130) | *I*w | ≡ | *w* |  |  |

ΞECREC(***σ***1*, g, A, I*) if *c* = 1 ΞSHA256(***σ***1*, g, A, I*) if *c* = 2 ΞRIP160(***σ***1*, g, A, I*) if *c* = 3 ΞID(***σ***1*, g, A, I*) if *c* = 4 ΞEXPMOD(***σ***1*, g, A, I*) if *c* = 5 ΞBN ADD(***σ***1*, g, A, I*) if *c* = 6 ΞBN MUL(***σ***1*, g, A, I*) if *c* = 7 ΞSNARKV(***σ***1*, g, A, I*) if *c* = 8 ΞBLAKE2 F(***σ***1*, g, A, I*) if *c* = 9

****

****

****

****

****

Ξ(***σ***1*, g, A, I*) otherwise

formal model of a virtual state machine, known as the Ethereum Virtual Machine (EVM). It is a *quasi-*Turing- complete machine; the *quasi* qualification comes from the fact that the computation is intrinsically bounded through a parameter, *gas*, which limits the total amount of compu- tation done.

* 1. **Basics.** The EVM is a simple stack-based architec- ture. The word size of the machine (and thus size of stack items) is 256-bit. This was chosen to facilitate the Keccak- 256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word- addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neu- mann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.

The machine can have exceptional execution for several reasons, including stack underflows and invalid instruc- tions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immedi- ately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

* 1. **Fees Overview.** Fees (denominated in gas) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation (see Appendix [G).](#_bookmark131) Secondly, gas may be deducted in order to form the payment for a subordinate

1. KEC(*I***b**) = ***σ***[*c*]c

It is assumed that the client will have stored the pair (KEC(*I***b**)*, I***b**) at some point prior in order to make the determination of *I***b** feasible.

As can be seen, there are nine exceptions to the usage of the general execution framework Ξ for evaluation of the message call: these are so-called ‘precompiled’ contracts, meant as a preliminary piece of architecture that may later become *native extensions*. The contracts in addresses 1 to 9 execute the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme, the identity function, arbitrary precision modular exponentiation, elliptic curve addition, elliptic curve scalar multiplication, an elliptic curve pairing check, and the BLAKE2 compression function F respectively. Their full formal definition is in Appendix [E.](#_bookmark119) We denote the set of the addresses of the precompiled contracts by *π*:

(133) *π* ≡ {1*,* 2*,* 3*,* 4*,* 5*,* 6*,* 7*,* 8*,* 9}

* 1. Execution Model

The execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a

message call or contract creation; this forms part of the payment for CREATE, CREATE2, CALL and CALLCODE. Finally, gas may be paid due to an increase in the usage of the memory.

Over an account’s execution, the total fee for memory- usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range. This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previ- ously indexed memory will certainly result in an additional memory usage fee. Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality. Storage fees have a slightly nuanced behaviour—to in- centivise minimisation of the use of storage (which corre- sponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given; in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more

than normal usage.

See Appendix [H](#_bookmark139) for a rigorous definition of the EVM gas cost.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 13

* 1. **Execution Environment.** In addition to the sys- tem state ***σ***, the remaining gas for computation *g*, and the accrued substate *A*, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple *I*:
     + *I*a, the address of the account which owns the code that is executing.
     + *I*o, the sender address of the transaction that orig- inated this execution.
     + *I*p, the price of gas in the transaction that origi- nated this execution.
     + *I***d**, the byte array that is the input data to this execution; if the execution agent is a transaction,

should halt.

(135) Ξ(***σ****, g, A, I*) ≡ (***σ****,* ***µ***g*, A,* **o**)

(136) (***σ****,* ***µ****, A, ...,* **o**) ≡ *X* (***σ****,* ***µ****, A, I*)

|  |  |  |  |
| --- | --- | --- | --- |
| (139) | ***µ*m** | ≡ | (0*,* 0*, ...*) |
| (140) | ***µ***i | ≡ | 0 |
| (141) | ***µ*s** | ≡ | () |
| (142) | ***µ*o** | ≡ | () |
| (143) |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| (137) | ***µ***g | ≡ | *g* |
| (138) | ***µ***pc | ≡ | 0 |

 ∅*,* ***µ****, A, I,* ∅ if *Z*(***σ****,* ***µ****, A, I*)

this would be the transaction data.

* + - *I*s, the address of the account which caused the

*X* (***σ****,* ***µ****, A, I*)

≡ **** ∅*,* ***µ****, A, I,* **o** if *w* =

REVERT

**** ·

code to be executing; if the execution agent is a transaction, this would be the transaction sender.

* + - *I*v, the value, in Wei, passed to this account as part of the same procedure as execution; if the

execution agent is a transaction, this would be the transaction value.

* + - *I***b**, the byte array that is the machine code to be

executed.

* + - *I*H, the block header of the present block.

where (144)

(145)

(146)

*O*(***σ****,* ***µ****, A, I*) **o** if **o** = ∅

*X O*(***σ****,* ***µ****, A, I*) otherwise

**o** ≡ *H*(***µ****, I*)

(*a, b, c, d*) · *e* ≡ (*a, b, c, d, e*)

***µ*** ≡ ***µ*** except:

* + - *I*e, the depth of the present message-call or

contract-creation (i.e. the number of CALLs or

CREATE(2)s being executed at present).

(147)

***µ***g

≡ ***µ***g

* *C*(***σ****,* ***µ****, A, I*)
  + *I*w, the permission to make modifications to the

state.

The execution model defines the function Ξ, which can compute the resultant state ***σ***, the remaining gas *g*, the resultant accrued substate *A* and the resultant output, **o**, given these definitions. For the present context, we will define it as:

(134) (***σ****, g, A,* **o**) ≡ Ξ(***σ****, g, A, I*)

where we will remember that *A*, the accrued substate, is defined in section [6.1.](#_bookmark36)

* 1. **Execution Overview.** We must now define the Ξ function. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, ***σ*** and the machine state, ***µ***. For- mally, we define it recursively with a function *X*. This uses an iterator function *O* (which defines the result of a

Note that, when we evaluate Ξ, we drop the fourth

element *I* and extract the remaining gas ***µ***g from the resultant machine state ***µ***.

*X* is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either [*Z*](#_bookmark59)becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until [*H*](#_bookmark60)becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

* + 1. *Machine State.* The machine state ***µ*** is defined as the tuple (*g, pc,* **m***, i,* **s**) which are the gas available, the

program counter *pc* ∈ N256 , the memory contents, the active number of words in memory (counting continuously

from position 0), and the stack contents. The memory contents ***µ*** are a series of zeroes of size 2256.

**m**

For the ease of reading, the instruction mnemonics,

written in small-caps (e.g. ADD), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given in Appendix [H.](#_bookmark139)

For the purposes of defining *Z*, *H* and *O*, we define *w*

as the current operation to be executed:

single cycle of the state machine) together with functions

[*Z*](#_bookmark59)which determines if the present state is an [exceptional](#_bookmark59)

(148) *w* ≡

f*I***b**[***µ***pc

] if ***µ***pc

*<* ll*I***b**ll

[halting](#_bookmark59) state of the machine and [*H*](#_bookmark60), specifying the output data of the instruction if and only if the present state is a [normal halting](#_bookmark60) state of the machine.

The empty sequence, denoted (), is not equal to the empty set, denoted ∅; this is important when interpreting the output of *H*, which evaluates to ∅ when execution is to continue but a series (potentially empty) when execution

[STOP](#_bookmark143) otherwise

We also assume the fixed amounts of *δ* and *α*, specifying the stack items removed and added, both subscriptable on the instruction and an instruction cost function *C* eval- uating to the full cost, in gas, of executing the given instruction.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 14

* + 1. *Exceptional Halting.* The exceptional halting func- tion *Z* is defined as:

(149)

*Z*(***σ****,* ***µ****, A, I*) ≡ ***µ****g < C*(***σ****,* ***µ****, A, I*) ∨

9.4.4. *Normal Halting.* The normal halting function *H* is defined:

(154)



*H*RETURN(***µ***) if *w* ∈ {[RETURN](#_bookmark149)*,* REVERT}

*δw* = ∅ ∨ ll***µ*s**ll *< δw* ∨

(*w* = JUMP ∧ ***µ*s**[0] ∈*/ D*(*I***b**)) ∨

*H*(***µ****, I*) ≡

() if *w* ∈ {[STOP](#_bookmark143)*,* [SELFDESTRUCT](#_bookmark152)}

∅ otherwise



where

(*w* = JUMPI ∧ ***µ*s**[1] = 0 ∧

***µ*s**[0] ∈*/ D*(*I***b**)) ∨

(*w* = RETURNDATACOPY ∧

***µ*s**[1] + ***µ*s**[2] *>* ll***µ*o**ll) ∨ ll***µ*s**ll − *δw* + *αw >* 1024 ∨ (¬*I*w ∧ *W* (*w,* ***µ***)) ∨

(*w* = SSTORE ∧ ***µ****g G*callstipend)

The data-returning halt operations, [RETURN](#_bookmark149) and REVERT, have a special function *H*RETURN. Note also the difference between the empty sequence and the empty set as discussed [here.](#_bookmark58)

* 1. **The Execution Cycle.** Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

(150) *W* (*w,* ***µ***) ≡ *w* ∈ {CREATE*,* CREATE2*,* SSTORE*,*

SELFDESTRUCT} ∨

LOG0 ≤ *w* ∧ *w* ≤ LOG4 ∨

*w* = CALL ∧ ***µ*s**[2] = 0

(155)

(156)

(157)

(158)

*O* (***σ****,* ***µ****, A, I*) ≡ (***σ****,* ***µ****, A, I*)

∆ ≡ *αw* − *δw*

ll***µ*s**ll ≡ ll***µ*s**ll + ∆

∀*x* ∈ [*αw,* ll***µ*s**ll) : ***µ*s**[*x*] ≡ ***µ*s**[*x* − ∆]

This states that the execution is in an exceptional halt- ing state if there is insufficient gas, if the instruction is invalid (and therefore its *δ* subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid, the new stack size would be larger than 1024 or state modification is attempted during a static call. The as-

The gas is reduced by the instruction’s gas cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function *J* , subscripted by one of two instructions, which evaluates to the according value:

tute reader will realise that this implies that no instruction

can, through its execution, cause an exceptional halt. Also, the execution is in an exceptional halting state if the gas

(159)

***µ***g ≡ ***µ***g − *C*(***σ****,* ***µ****, A, I*)

[*J*JUMP](#_bookmark147)(***µ***) if *w* = JUMP

left prior to executing an [SSTORE](#_bookmark145) instruction is less than or equal to the call stipend [*G*callstipend](#_bookmark134). The last condition was introduced in EIP-1706 by [Forshtat and Weiss [2019]](#_bookmark96)

(160)

***µ***pc ≡

[*J*JUMPI](#_bookmark148)(***µ***) if *w* = JUMPI

*N* (***µ***pc*, w*) otherwise

(part of EIP-2200 by [Tang](#_bookmark105) [[2019]).](#_bookmark105)

In general, we assume the memory, accrued substate

and system state do not change:

* + 1. *Jump Destination Validity.* We previously used *D* as the function to determine the set of valid jump desti- nations given the code that is being run. We define this as any position in the code occupied by a JUMPDEST instruction.

(161)

(162)

(163)

(164)

***µ*m** ≡ ***µ*m *µ***i ≡ ***µ***i

*A* ≡ *A*

***σ*** ≡ ***σ***

All such positions must be on valid instruction bound- aries, rather than sitting in the data portion of PUSH operations and must appear within the explicitly defined portion of the code (rather than in the implicitly defined [STOP](#_bookmark143) operations that trail it).

Formally:

(151) *D*(**c**) ≡ *D*J(**c***,* 0) where:

(152)

****

{} if *i* ;;: ll**c**ll

*D* (**c***, i*) {*i*} ∪ *D*J(**c***, N* (*i,* **c**[*i*]))

J

≡

if **c**[*i*] = JUMPDEST

****



*D*J(**c***, N* (*i,* **c**[*i*])) otherwise

where *N* is the next valid instruction position in the code, skipping the data of a PUSH instruction, if any: (153)



*i* + *w* − PUSH1 + 2

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix [H,](#_bookmark139) alongside values for *α* and *δ* and a formal description of the gas requirements.

* 1. Blocktree to Blockchain

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the *heaviest* path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined [genesis](#_bookmark153) [block,](#_bookmark153) in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols.

Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any

*N* (*i, w*) ≡

∈

if *w* [PUSH1*,* PUSH32]

*i* + 1 otherwise

block contributes toward the total computation or *total difficulty* of a chain.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 15

as:

Thus we define the total difficulty of block *B* recursively

If there are collisions of the beneficiary addresses be- tween ommers and the block (i.e. two ommers with the

(165)

(166)

*B*t ≡ *B*t + *B*d *B* ≡ *P* (*B*H)

same beneficiary address or an ommer with the same bene- ficiary address as the present block), additions are applied cumulatively.

As such given a block *B*, *B*t is its total difficulty, *B* is its parent block and *B*d is its difficulty.



We define the block reward in Wei: (176)

1. Block Finalisation

The process of finalising a block involves four stages:

* + - 1. Validate (or, if mining, determine) ommers;

*R*block = 1018 ×

5 if *H*i *< F*Byzantium

3 if *F*Byzantium *H*i *< F*Constantinople

2 if *H*i ;;: *F*Constantinople

* + - 1. validate (or, if mining, determine) transactions;
      2. apply rewards;
      3. verify (or, if mining, compute a valid) state and [block nonce.](#_bookmark17)

11.1. **Ommer Validation.** The validation of [ommer](#_bookmark18) [headers](#_bookmark18) means nothing more than verifying that each om- mer header is both a valid header and satisfies the relation

of *N* th-generation ommer to the present block where *N* ≤ 6.

The maximum of ommer headers is two. Formally:

11.4. **State & Nonce Validation.** We may now define

the function, Γ, that maps a block *B* to its initiation state: (177)

f≡

Γ(*B*) ***σ***0 if *P* (*B*H) = ∅

***σ***i : [TRIE](#_bookmark118)(*L*S(***σ***i)) = *P* (*B*H)Hr otherwise

Here, TRIE(*L*S(***σ***i)) means the hash of the root node of a trie of state ***σ***i; it is assumed that implementations will store this in the state database, which is trivial and efficient

since the trie is by nature an immutable data structure.

(167) ll*B***U**

ll 2

**U**∈*B***U**

[*V* (**U**](#_bookmark35)) ∧ *k*(**U***, P* (**B**

**H**)**H**

*,* 6)

And finally we define Φ, the block transition function, which maps an incomplete block *B* to a complete block *B*:

where *k* denotes the “is-kin” property:

(168)



(178) Φ(*B*) ≡ *B* : *B* = *B*∗ except:

*k*(*U, H, n*) ≡

*false* if *n* = 0

*s*(*U, H*)

(179)

*B*n = *n* : *x*

2256

[*H*d](#_bookmark31)

 ∨ *k*(*U, P* (*H*)H*, n* − 1) otherwise

and *s* denotes the “is-sibling” property:

(180) *B*m = *m* with (*x, m*) = PoW(*B*[n∗](#_bookmark34) *, n,* **d**)

(181)

*B*∗ ≡ *B* except: [*B*r∗](#_bookmark21) = [*r*](#_bookmark128)[(Π(Γ(](#_bookmark67)*B*)*, B*))

(169)

*s*(*U, H*) ≡ (*P* (*H*) = *P* (*U* ) ∧ *H U* ∧ *U* ∈*/ B*(*H*)**U**)

where *B*(*H*) and *P* (*H*) are the block and the parent block of the corresponding header *H* respectively.

* 1. **Transaction Validation.** The given **gasUsed** must correspond faithfully to the transactions listed: [*B*Hg](#_bookmark14), the total gas used in the block, must be equal to the accumulated gas used according to the final transaction:

(170) [*B*H](#_bookmark14)g = [*.€*](#_bookmark2)([**R**)u](#_bookmark22)

* 1. **Reward Application.** The application of rewards to a block involves raising the balance of the accounts of the beneficiary address of the block and each ommer by a certain amount. We raise the block’s beneficiary account by [*R*block](#_bookmark62); for each ommer, we raise the block’s beneficiary by an additional 1 of the block reward and the benefi- ciary of the ommer gets rewarded depending on the block number. Formally we define the function Ω:

32

(171)

Ω(*B,* ***σ***) ≡ ***σ*** : ***σ*** = ***σ*** except:

With **d** being a dataset as specified in Appendix [J.](#_bookmark154)

As specified at the beginning of the present work, [Π](#_bookmark67) is the state-transition function, which is defined in terms of [Ω,](#_bookmark65) the block finalisation function and [Υ,](#_bookmark0) the transaction- evaluation function, both now well-defined.

As previously detailed, **R**[*n*]z, **R**[*n*]**l** and **R**[*n*]u are the *n*th corresponding status code, logs and cumulative gas used after each transaction (**R**[*n*]b, the fourth component in the tuple, has already been defined in terms of the logs). We also define the *n*th state ***σ***[*n*], which is defined simply as the state resulting from applying the corresponding

transaction to the state resulting from the previous trans- action (or the block’s initial state in the case of the first such transaction):

f

(182) ***σ***[*n*] = [Γ(](#_bookmark64)*B*) if *n <* 0

[Υ(](#_bookmark0)***σ***[*n* − 1]*, B***T**[*n*]) otherwise

In the case of *B***R**[*n*]u, we take a similar approach defin- ing each item as the gas used in evaluating the correspond- ing transaction summed with the previous item (or zero, if it is the first), giving us a running total:

(172) ***σ*** [**B** ] = ***σ***[**B** ] + (1 + ll**BU**ll *R*

Hc b

Hc b

32

block

(183) **R**[*n*]

=

Υ*g*(***σ***[*n* − 1]*, B*

[*n*])

0 if *n <* 0

(173)

u

**T**

∀**U** ∈ **BU** :

 +**R**[*n* − 1]u otherwise

f∅ if ***σ***[**U** ] = ∅ ∧ *R* = 0

For **R**[*n*]**l**, we utilise the [Υ](#_bookmark42) function that we conve-

***σ*** [**U**c] =

[**l**](#_bookmark42)

c

**a** otherwise

niently defined in the transaction execution function.

(184) **R**[*n*]**l** = Υ**l**(***σ***[*n* − 1]*, B***T**[*n*])

(174) **a** ≡ (***σ***[*U*c]n*,* ***σ***[*U*c]b + *R,* ***σ***[*U*c]**s***,* ***σ***[*U*c]c)

(175) *R* ≡ (1 + 1 (*U* − *B* ) *R*

8 i Hi

block

We define [**R**[*n*]z](#_bookmark42) in a similar manner.

(185) **R**[*n*]z = Υz(***σ***[*n* − 1]*, B***T**[*n*])

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 16

Finally, we define Π as the new state given the block re- ward function [Ω](#_bookmark65) applied to the final transaction’s resultant state, [*.€*](#_bookmark2)(***σ***):

(186) Π(***σ****, B*) ≡ Ω(*B, .€*(***σ***))

Thus the complete block-transition mechanism is de- fined, except for PoW, the proof-of-work function.

11.5. **Mining Proof-of-Work.** The mining proof-of- work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the deter- mination of some token value *n*. It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total dif- ficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism.

For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as possible, and, ideally, makes the act of mining a simple swap from electricity to Ether at roughly the same rate for anyone around the world.

Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network’s total mining power and as such gives them a super-linear reward (thus skewing distribution in their favour) as well as reducing the network security.

One plague of the Bitcoin world is ASICs. These are specialised pieces of compute hardware that exist only to do a single task [(Smith [1997]).](#_bookmark98) In Bitcoin’s case the task is the SHA256 hash function [(Courtois et al. [2014]).](#_bookmark84) While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work func- tion that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the [nonce](#_bookmark17) requires a lot of mem- ory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of “specialised hardware” for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to “specialised hardware” for the task. For Ethereum 1.0 we have chosen the first path.

More formally, the proof-of-work function takes the form of PoW:

(187)

256

2

*m* = [*H*m](#_bookmark16) ∧ *n* [*H*](#_bookmark31)with (*m, n*) = PoW([*H*n](#_bookmark34)*, H*n*,* **d**)

[d](#_bookmark31)

Where *H*n is the new block’s header but *without* the nonce and mix-hash components; *H*n is the nonce of the header; **d** is a large data set needed to compute the mix- Hash and *H*d is the new block’s difficulty value (i.e. the

block difficulty from section [10).](#_bookmark61) PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudo- random number cryptographically dependent on *H* and **d**. The underlying algorithm is called Ethash and is described below.

11.5.1. *Ethash.* Ethash is the PoW algorithm for Ethereum

1.0. It is the latest version of Dagger-Hashimoto, intro- duced by [Buterin](#_bookmark78) [[2013b]](#_bookmark78) and [Dryja](#_bookmark90) [[2014],](#_bookmark90) although it can no longer appropriately be called that since many of the original features of both algorithms were drastically changed with R&D from February 2015 until May 4 2015 [(Jentzsch](#_bookmark106) [[2015]).](#_bookmark106) The general route that the algorithm takes is as follows:

There exists a seed which can be computed for each block by scanning through the block headers up until that point. From the seed, one can compute a pseudorandom cache, [*J*cacheinit](#_bookmark155) bytes in initial size. Light clients store the cache. From the cache, we can generate a dataset, [*J*datasetinit](#_bookmark156) bytes in initial size, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache. The large dataset is updated once every

[*J*epoch](#_bookmark157) blocks, so the vast majority of a miner’s effort will be reading the dataset, not making changes to it. The

mentioned parameters as well as the algorithm is explained in detail in Appendix [J.](#_bookmark154)

1. Implementing Contracts

There are several patterns of contracts engineering that allow particular useful behaviours; two of these that we will briefly discuss are data feeds and random numbers.

* 1. **Data Feeds.** A data feed contract is one which pro- vides a single service: it gives access to information from the external world within Ethereum. The accuracy and timeliness of this information is not guaranteed and it is the task of a secondary contract author—the contract that utilises the data feed—to determine how much trust can be placed in any single data feed.

The general pattern involves a single contract within Ethereum which, when given a message call, replies with some timely information concerning an external phenome- non. An example might be the local temperature of New York City. This would be implemented as a contract that returned that value of some known point in storage. Of course this point in storage must be maintained with the correct such temperature, and thus the second part of the pattern would be for an external server to run an Ethereum node, and immediately on discovery of a new block, creates a new valid transaction, sent to the contract, updating said value in storage. The contract’s code would accept such updates only from the identity contained on said server.

* 1. **Random Numbers.** Providing random numbers within a deterministic system is, naturally, an impossible task. However, we can approximate with pseudo-random numbers by utilising data which is generally unknowable

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 17

at the time of transacting. Such data might include the block’s hash, the block’s timestamp and the block’s benefi- ciary address. In order to make it hard for malicious miners to control those values, one should use the [BLOCKHASH](#_bookmark144) operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result.

1. Future Directions

The state database won’t be forced to maintain all past state [trie](#_bookmark118) structures into the future. It should maintain an age for each node and eventually discard nodes that are neither recent enough nor checkpoints. Checkpoints, or a set of nodes in the database that allow a particular block’s state trie to be traversed, could be used to place a maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain.

Blockchain consolidation could be used in order to re- duce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 10,000th block) could be maintained by the peer network, effectively recasting the [genesis block.](#_bookmark153) This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks.

Finally, blockchain compression could perhaps be con- ducted: nodes in state trie that haven’t sent/received a transaction in some constant amount of blocks could be thrown out, reducing both Ether-leakage and the growth of the state database.

13.1. **Scalability.** Scalability remains an eternal concern. With a generalised state transition function, it becomes dif- ficult to partition and parallelise transactions to apply the divide-and-conquer strategy. Unaddressed, the dynamic value-range of the system remains essentially fixed and as the average transaction value increases, the less valuable of them become ignored, being economically pointless to in- clude in the main ledger. However, several strategies exist that may potentially be exploited to provide a considerably more scalable protocol.

Some form of hierarchical structure, achieved by either consolidating smaller lighter-weight chains into the main block or building the main block through the incremen- tal combination and adhesion (through proof-of-work) of smaller transaction sets may allow parallelisation of trans- action combination and block-building. Parallelism could also come from a prioritised set of parallel blockchains, consolidating each block and with duplicate or invalid transactions thrown out accordingly.

Finally, verifiable computation, if made generally avail- able and efficient enough, may provide a route to allow the proof-of-work to be the verification of final state.

1. Conclusion

We have introduced, discussed and formally defined the protocol of Ethereum. Through this protocol the reader may implement a node on the Ethereum network and join others in a decentralised secure social operating system. Contracts may be authored in order to algorithmically specify and autonomously enforce rules of interaction.

1. Acknowledgements

Many thanks to Aeron Buchanan for authoring the *Homestead* revisions, Christoph Jentzsch for authoring the Ethash algorithm and Yoichi Hirai for doing most of the EIP-150 changes. Important maintenance, useful correc- tions and suggestions were provided by a number of others from the Ethereum DEV organisation and Ethereum com- munity at large including Gustav Simonsson, Pawelw Bylica, Jutta Steiner, Nick Savers, Viktor Tr´on, Marko Simovic, Giacomo Tazzari and, of course, Vitalik Buterin.

1. Availability

The source of this paper is maintained at [https:](https://github.com/ethereum/yellowpaper/)

[//github.com/ethereum/yellowpaper/](https://github.com/ethereum/yellowpaper/). An auto- generated PDF is located at [https://ethereum.github.](https://ethereum.github.io/yellowpaper/paper.pdf) [io/yellowpaper/paper.pdf](https://ethereum.github.io/yellowpaper/paper.pdf).

References

Jacob Aron. BitCoin software finds new life. *New Scientist*, 213(2847):20, 2012. URL [http://www.sciencedirect.](http://www.sciencedirect.com/science/article/pii/S0262407912601055) [com/science/article/pii/S0262407912601055](http://www.sciencedirect.com/science/article/pii/S0262407912601055).

Adam Back. Hashcash - Amortizable Publicly Auditable

Cost-Functions, 2002. URL [http://www.hashcash.org/](http://www.hashcash.org/papers/amortizable.pdf) [papers/amortizable.pdf](http://www.hashcash.org/papers/amortizable.pdf).

Tim Beiko, James Hancock, and Thomas Jay Rush. EIP- 4345: Difficulty bomb delay to June 2022, October 2021a. URL <https://eips.ethereum.org/EIPS/eip-4345>.

Tim Beiko et al. Berlin network upgrade specifica-

tion, 2021b. URL [https://github.com/ethereum/](https://github.com/ethereum/eth1.0-specs/blob/master/network-upgrades/mainnet-upgrades/berlin.md) [eth1.0-specs/blob/master/network-upgrades/](https://github.com/ethereum/eth1.0-specs/blob/master/network-upgrades/mainnet-upgrades/berlin.md) [mainnet-upgrades/berlin.md](https://github.com/ethereum/eth1.0-specs/blob/master/network-upgrades/mainnet-upgrades/berlin.md).

Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. The KECCAK SHA-3 sub- mission, 2011. URL [https://keccak.team/files/](https://keccak.team/files/Keccak-submission-3.pdf)

[Keccak-submission-3.pdf](https://keccak.team/files/Keccak-submission-3.pdf).

Roman Boutellier and Mareike Heinzen. Pirates, Pio- neers, Innovators and Imitators. In *Growth Through Innovation*, pages 85–96. Springer, 2014. URL [https:](https://www.springer.com/gb/book/9783319040158)

[//www.springer.com/gb/book/9783319040158](https://www.springer.com/gb/book/9783319040158).

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform,

2013a. URL [https://github.com/ethereum/wiki/](https://github.com/ethereum/wiki/wiki/White-Paper) [wiki/White-Paper](https://github.com/ethereum/wiki/wiki/White-Paper).

Vitalik Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative, 2013b. URL <http://www.hashcash.org/papers/dagger.html>.

Vitalik Buterin. EIP-2: Homestead hard-fork changes,

2015. URL <https://eips.ethereum.org/EIPS/eip-2>. Vitalik Buterin. EIP-100: Change difficulty adjustment to target mean block time including uncles, April 2016a.

URL <https://eips.ethereum.org/EIPS/eip-100>. Vitalik Buterin. EIP-155: Simple replay attack protec-

tion, October 2016b. URL [https://eips.ethereum.](https://eips.ethereum.org/EIPS/eip-155) [org/EIPS/eip-155](https://eips.ethereum.org/EIPS/eip-155).

Vitalik Buterin. EIP-1014: Skinny CREATE2, April 2018. URL <https://eips.ethereum.org/EIPS/eip-1014>.

Vitalik Buterin and Martin Swende. EIP-2929: Gas cost in-

creases for state access opcodes, September 2020a. URL

<https://eips.ethereum.org/EIPS/eip-2929>.

Vitalik Buterin and Martin Swende. EIP-2930: Optional ac- cess lists, August 2020b. URL [https://eips.ethereum.](https://eips.ethereum.org/EIPS/eip-2930) [org/EIPS/eip-2930](https://eips.ethereum.org/EIPS/eip-2930).

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 18

Eric Conner. EIP-2384: Muir Glacier difficulty bomb delay, November 2019. URL [https://eips.ethereum.](https://eips.ethereum.org/EIPS/eip-2384) [org/EIPS/eip-2384](https://eips.ethereum.org/EIPS/eip-2384).

Nicolas T. Courtois, Marek Grajek, and Rahul Naik.

*Optimizing SHA256 in Bitcoin Mining*, pages 131–

144. Springer Berlin Heidelberg, Berlin, Heidel- berg, 2014. ISBN 978-3-662-44893-9. doi: 10. 1007/978-3-662-44893-9 12. URL [https://doi.org/10.](https://doi.org/10.1007/978-3-662-44893-9_12) [1007/978-3-662-44893-9\_12](https://doi.org/10.1007/978-3-662-44893-9_12).

B.A. Davey and H.A. Priestley. *Introduction to lattices and order. 2nd ed.* Cambridge: Cambridge University Press, 2nd ed. edition, 2002. ISBN 0-521-78451-4/pbk. Thaddeus Dryja. Hashimoto: I/O bound proof of work, 2014. URL [http://diyhpl.us/~bryan/papers2/](http://diyhpl.us/~bryan/papers2/bitcoin/meh/hashimoto.pdf)

[bitcoin/meh/hashimoto.pdf](http://diyhpl.us/~bryan/papers2/bitcoin/meh/hashimoto.pdf).

Cynthia Dwork and Moni Naor. Pricing via pro- cessing or combatting junk mail. In *In 12th An- nual International Cryptology Conference*, pages 139– 147, 1992. URL [http://www.wisdom.weizmann.ac.il/](http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf)

[~naor/PAPERS/pvp.pdf](http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf).

Dankrad Feist, Dmitry Khovratovich, and Marius van der

Wijden. EIP-3607: Reject transactions from senders with deployed code, June 2021. URL [https://eips.](https://eips.ethereum.org/EIPS/eip-3607) [ethereum.org/EIPS/eip-3607](https://eips.ethereum.org/EIPS/eip-3607).

Alex Forshtat and Yoav Weiss. EIP-1706: Disable sstore

with gasleft lower than call stipend, 2019. URL [https:](https://eips.ethereum.org/EIPS/eip-1706)

[//eips.ethereum.org/EIPS/eip-1706](https://eips.ethereum.org/EIPS/eip-1706).

Phong Vo Glenn Fowler, Landon Curt Noll. FowlerNol- lVo hash function, 1991. URL [http://www.isthe.com/](http://www.isthe.com/chongo/tech/comp/fnv/index.html) [chongo/tech/comp/fnv/index.html](http://www.isthe.com/chongo/tech/comp/fnv/index.html).

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle,

and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004. URL [https://www.iacr.org/](https://www.iacr.org/archive/ches2004/31560117/31560117.pdf) [archive/ches2004/31560117/31560117.pdf](https://www.iacr.org/archive/ches2004/31560117/31560117.pdf).

James Hancock. EIP-3554: Difficulty bomb delay to De- cember 2021, May 2021. URL [https://eips.ethereum.](https://eips.ethereum.org/EIPS/eip-3554) [org/EIPS/eip-3554](https://eips.ethereum.org/EIPS/eip-3554).

Tjaden Hess, Matt Luongo, Piotr Dyraga, and James

Hancock. EIP-152: Add BLAKE2 compression func- tion ‘F‘ precompile, October 2016. URL [https://eips.](https://eips.ethereum.org/EIPS/eip-152) [ethereum.org/EIPS/eip-152](https://eips.ethereum.org/EIPS/eip-152).

Christoph Jentzsch. Commit date for ethash, 2015. URL

[https://github.com/ethereum/yellowpaper/commit/](https://github.com/ethereum/yellowpaper/commit/77a8cf2428ce245bf6e2c39c5e652ba58a278666#commitcomment-24644869) [77a8cf2428ce245bf6e2c39c5e652ba58a278666#](https://github.com/ethereum/yellowpaper/commit/77a8cf2428ce245bf6e2c39c5e652ba58a278666#commitcomment-24644869) [commitcomment-24644869](https://github.com/ethereum/yellowpaper/commit/77a8cf2428ce245bf6e2c39c5e652ba58a278666#commitcomment-24644869).

Don Johnson, Alfred Menezes, and Scott Van- stone. The Elliptic Curve Digital Signa- ture Algorithm (ECDSA), 2001. URL https:

//web.archive.org/web/20170921160141/http://

cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf. Accessed 21 September 2017, but the original link was

inaccessible on 19 October 2017. Refer to section 6.2 for ECDSAPUBKEY, and section 7 for ECDSASIGN and

ECDSARECOVER.

Sergio Demian Lerner. Strict Memory Hard Hashing Func- tions, 2014. URL [http://www.hashcash.org/papers/](http://www.hashcash.org/papers/memohash.pdf) [memohash.pdf](http://www.hashcash.org/papers/memohash.pdf).

Mark Miller. The Future of Law. In *pa-*

*per delivered at the Extro 3 Conference (August 9)*, 1997. URL [https://drive.google.com/file/d/](https://drive.google.com/file/d/0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing) [0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing](https://drive.google.com/file/d/0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing).

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash

system, 2008. URL [http://www.bitcoin.org/bitcoin.](http://www.bitcoin.org/bitcoin.pdf) [pdf](http://www.bitcoin.org/bitcoin.pdf).

Meni Rosenfeld, Yoni Assia, Vitalik Buterin, m li- orhakiLior, Oded Leiba, Assaf Shomer, and Eli- ran Zach. Colored Coins Protocol Specification, 2012. URL [https://github.com/Colored-Coins/](https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification)

[Colored-Coins-Protocol-Specification](https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification).

Markku-Juhani Saarinen and Jean-Philippe Aumasson. RFC 7693: The BLAKE2 cryptographic hash and mes- sage authentication code (MAC), November 2015. URL <https://tools.ietf.org/html/rfc7693>.

Afri Schoedon. EIP-1234: Constantinople difficulty bomb

delay and block reward adjustment, 2018. URL [https:](https://eips.ethereum.org/EIPS/eip-1234)

[//eips.ethereum.org/EIPS/eip-1234](https://eips.ethereum.org/EIPS/eip-1234).

Afri Schoedon and Vitalik Buterin. EIP-649: Metropo- lis difficulty bomb delay and block reward reduction, June 2017. URL [https://eips.ethereum.org/EIPS/](https://eips.ethereum.org/EIPS/eip-649)

[eip-649](https://eips.ethereum.org/EIPS/eip-649).

Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997. ISBN 0201500221.

Yonatan Sompolinsky and Aviv Zohar. Accelerating bit- coin’s transaction processing. fast money grows on trees, not chains, 2013. URL [https://eprint.iacr.org/](https://eprint.iacr.org/2013/881)

[2013/881](https://eprint.iacr.org/2013/881).

Simon Sprankel. Technical Basis of Digital Currencies, 2013. URL [http://www.coderblog.de/wp-content/](http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf) [uploads/technical-basis-of-digital-currencies.](http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf) [pdf](http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf).

Nick Szabo. Formalizing and securing relation-

ships on public networks. *First Monday*, 2(9), 1997. URL [http://firstmonday.org/ojs/index.php/](http://firstmonday.org/ojs/index.php/fm/article/view/548) [fm/article/view/548](http://firstmonday.org/ojs/index.php/fm/article/view/548).

Wei Tang. EIP-2200: Structured definitions for net gas me-

tering, 2019. URL [https://eips.ethereum.org/EIPS/](https://eips.ethereum.org/EIPS/eip-2200) [eip-2200](https://eips.ethereum.org/EIPS/eip-2200).

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gn Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing, 2003. URL [https://](https://www.cs.cornell.edu/people/egs/papers/karma.pdf)

[www.cs.cornell.edu/people/egs/papers/karma.pdf](https://www.cs.cornell.edu/people/egs/papers/karma.pdf).

J. R. Willett. MasterCoin Complete Specification, 2013. URL <https://github.com/mastercoin-MSC/spec>.

Micah Zoltu. EIP-2718: Typed transaction envelope,

June 2020. URL [https://eips.ethereum.org/EIPS/](https://eips.ethereum.org/EIPS/eip-2718) [eip-2718](https://eips.ethereum.org/EIPS/eip-2718).

Appendix A. Terminology

**External Actor:** A person or other entity able to interface to an Ethereum node, but external to the world of Ethereum. It can interact with Ethereum through depositing signed Transactions and inspecting the blockchain and associated state. Has one (or more) intrinsic Accounts.

**Address:** A 160-bit code used for identifying Accounts.

**Account:** Accounts have an intrinsic balance and transaction count maintained as part of the Ethereum state. They also have some (possibly empty) EVM Code and a (possibly empty) Storage State associated with them.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 19

Though homogenous, it makes sense to distinguish between two practical types of account: those with empty associated EVM Code (thus the account balance is controlled, if at all, by some external entity) and those with non-empty associated EVM Code (thus the account represents an Autonomous Object). Each Account has a single Address that identifies it.

**Transaction:** A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain.

**Autonomous Object:** A notional object existent only within the hypothetical state of Ethereum. Has an intrinsic address and thus an associated account; the account will have non-empty associated EVM Code. Incorporated only as the Storage State of that account.

**Storage State:** The information particular to a given Account that is maintained between the times that the Account’s associated EVM Code runs.

**Message:** Data (as a set of bytes) and Value (specified as Ether) that is passed between two Accounts, either through the deterministic operation of an Autonomous Object or the cryptographically secure signature of the Transaction.

**Message Call:** The act of passing a message from one Account to another. If the destination account is associated with non-empty EVM Code, then the VM will be started with the state of said Object and the Message acted upon. If the message sender is an Autonomous Object, then the Call passes any data returned from the VM operation.

**Gas:** The fundamental network cost unit. Paid for exclusively by Ether (as of PoC-4), which is converted freely to and from Gas as required. Gas does not exist outside of the internal Ethereum computation engine; its price is set by the Transaction and miners are free to ignore Transactions whose Gas price is too low.

**Contract:** Informal term used to mean both a piece of EVM Code that may be associated with an Account or an Autonomous Object.

**Object:** Synonym for Autonomous Object.

**App:** An end-user-visible application hosted in the Ethereum Browser.

**Ethereum Browser:** (aka Ethereum Reference Client) A cross-platform GUI of an interface similar to a simplified browser (a la Chrome) that is able to host sandboxed applications whose backend is purely on the Ethereum protocol.

**Ethereum Virtual Machine:** (aka EVM) The virtual machine that forms the key part of the execution model for an Account’s associated EVM Code.

**Ethereum Runtime Environment:** (aka ERE) The environment which is provided to an Autonomous Object executing in the EVM. Includes the EVM but also the structure of the world state on which the EVM relies for certain I/O instructions including CALL & CREATE.

**EVM Code:** The bytecode that the EVM can natively execute. Used to formally specify the meaning and ramifications of a message to an Account.

**EVM Assembly:** The human-readable form of EVM-code.

**LLL:** The Lisp-like Low-level Language, a human-writable language used for authoring simple contracts and general low-level language toolkit for trans-compiling to.

Appendix B. Recursive Length Prefix

This is a serialisation method for encoding arbitrarily structured binary data (byte arrays). We define the set of possible structures T:

(188)

(189)

(190)

T ≡ L \± B

L ≡ {**t** : **t** = (**t**[0]*,* **t**[1]*, ...*) ∧ ∀*n <* ll**t**ll : **t**[*n*] ∈ T}

B ≡ {**b** : **b** = (**b**[0]*,* **b**[1]*, ...*) ∧ ∀*n <* ll**b**ll : **b**[*n*] ∈ O}

Where O is the set of (8-bit) bytes. Thus B is the set of all sequences of bytes (otherwise known as byte arrays, and a leaf if imagined as a tree), L is the set of all tree-like (sub-)structures that are not a single leaf (a branch node if imagined as a tree) and T is the set of all byte arrays and such structural sequences. The disjoint union \± is needed only

to distinguish the empty byte array () ∈ B from the empty list () ∈ L, which are encoded differently as defined below; as

common, we will abuse notation and leave the disjoint union indices implicit, inferable from context.

We define the RLP function as RLP through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

≡ f

(191) RLP(**x**) *R*b(**x**) if **x** ∈ B

*R*l(**x**) otherwise

If the value to be serialised is a byte array, the RLP serialisation takes one of three forms:

* If the byte array contains solely a single byte and that single byte is less than 128, then the input is exactly equal to the output.
* If the byte array contains fewer than 56 bytes, then the output is equal to the input prefixed by the byte equal to the length of the byte array plus 128.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 20

* Otherwise, the output is equal to the input, provided that it contains fewer than 2 bytes, prefixed by the

64

minimal-length byte array which when interpreted as a big-endian integer is equal to the length of the input byte

array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Byte arrays containing 264 or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding of a byte array is always below 192, and thus it can be readily distinguished from the encodings of sequences in L.

Formally, we define *R*b:

(192)

*R*b(**x**) ≡

**x** if ll**x**ll = 1 ∧ **x**[0] *<* 128

(128 + ll**x**ll) · **x** else if ll**x**ll *<* 56

**** 183 + BE(ll**x**ll) · BE(ll**x**ll) · **x** else if ll**x**ll *<* 264

****

∅ otherwise



**b** −1

(193)

BE(*x*) ≡ (*b*0*, b*1*, ...*) : *b*0

0 ∧ *x* =

*n* =0

*bn* · 256 **b** −1−*n*

(194) (*x*1*, ..., xn*) · (*y*1*, ..., ym*) = (*x*1*, ..., xn, y*1*, ..., ym*)

Thus BE is the function that expands a non-negative integer value to a big-endian byte array of minimal length and the dot operator performs sequence concatenation.

If instead, the value to be serialised is a sequence of other items then the RLP serialisation takes one of two forms:

* If the concatenated serialisations of each contained item is less than 56 bytes in length, then the output is equal to that concatenation prefixed by the byte equal to the length of this byte array plus 192.
* Otherwise, the output is equal to the concatenated serialisations, provided that they contain fewer than 2 bytes,

64

prefixed by the minimal-length byte array which when interpreted as a big-endian integer is equal to the length

of the concatenated serialisations byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 247.

Sequences whose concatenated serialized items contain 264 or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding does not exceed 255 (otherwise it would not be a byte).

Thus we finish by formally defining *R*l:

(192 + ll*s*(**x**)ll) · *s*(**x**) if *s*(**x**) = ∅ ∧ ll*s*(**x**)ll *<* 56

(195)

(196)

*R*l(**x**) ≡

*s*(**x**) ≡

247 + BE(ll*s*(**x**)ll) · BE(ll*s*(**x**)ll) · *s*(**x**) else if *s*(**x**) = ∅ ∧ ll*s*(**x**)ll *<* 264

f

∅ otherwise

∅

RLP(**x**[0]) · RLP(**x**[1]) · *...* if ∀*i* : RLP(**x**[*i*])

∅ otherwise

If RLP is used to encode a scalar, defined only as a non-negative integer (in N, or in N*x* for any *x*), it must be encoded as the shortest byte array whose big-endian interpretation is the scalar. Thus the RLP of some non-negative integer *i* is defined as:

(197) RLP(*i* : *i* ∈ N) ≡ RLP(BE(*i*))

When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

There is no specific canonical encoding format for signed or floating-point values.

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

It is defined as the function HP which maps from a sequence of nibbles (represented by the set Y) together with a boolean value to a sequence of bytes (represented by the set B):

(198)

HP(**x***, t*) : **x** Y (16*f* (*t*)*,* 16**x**[0] + **x**[1]*,* 16**x**[2] + **x**[3]*, ...*) if ll**x**ll is even

(16(*f* (*t*) + 1) + **x**[0]*,* 16**x**[1] + **x**[2]*,* 16**x**[3] + **x**[4]*, ...*) otherwise

f∈ ≡

(199)

*f* (*t*) ≡ f2 if *t* 0

0 otherwise

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag *t*. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 21

Appendix D. Modified Merkle Patricia Tree

The modified Merkle Patricia tree (trie) provides a persistent data structure to map between arbitrary-length binary data (byte arrays). It is defined in terms of a mutable data structure to map between 256-bit binary fragments and arbitrary-length binary data, typically implemented as a database. The core of the trie, and its sole requirement in terms of the protocol specification, is to provide a single value that identifies a given set of key-value pairs, which may be either a 32-byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner that allows effective and efficient realisation of the protocol.

Formally, we assume the input value I, a set containing pairs of byte sequences with unique keys:

(200) I = {(**k**0 ∈ B*,* **v**0 ∈ B)*,* (**k**1 ∈ B*,* **v**1 ∈ B)*,* }

When considering such a sequence, we use the common numeric subscript notation to refer to a tuple’s key or value, thus:

(201) ∀*I* ∈ I : *I* ≡ (*I*0*, I*1)

Any series of bytes may also trivially be viewed as a series of nibbles, given an endian-specific notation; here we assume big-endian. Thus:

(202)

(203)

∀*n* : ∀*i <* 2ll**k***n*

*y*(I) = {(**k**0 ∈ Y*,* **v**0 ∈ B)*,* (**k**1 ∈ Y*,* **v**1 ∈ B)*, ...*}

**k***n*[*i* 2] 16 if *i* is even

fl ÷ ÷ Jll : **k** [*i*] ≡

*n*

**k***n*[l*i* ÷ 2J] mod 16 otherwise

We define the function TRIE, which evaluates to the root of the trie that represents this set when encoded in this structure:

(204) TRIE(I) ≡ KEC RLP(*c*(I*,* 0))

We also assume a function *n*, the trie’s node cap function. When composing a node, we use RLP to encode the structure. As a means of reducing storage complexity, we store nodes whose composed RLP is fewer than 32 bytes directly; for those larger we assert prescience of the byte array whose Keccak-256 hash evaluates to our reference. Thus we define in terms of *c*, the node composition function:

() if I = ∅

(205) *n*(I*, i*) ≡ *c*(I*, i*) if ll RLP *c*(I*, i*) ll *<* 32



KEC RLP(*c*(I*, i*)) otherwise



In a manner similar to a radix tree, when the trie is traversed from root to leaf, one may build a single key-value pair. The key is accumulated through the traversal, acquiring a single nibble from each branch node (just as with a radix tree). Unlike a radix tree, in the case of multiple keys sharing the same prefix or in the case of a single key having a unique suffix, two optimising nodes are provided. Thus while traversing, one may potentially acquire multiple nibbles from each of the other two node types, extension and leaf. There are three kinds of nodes in the trie:

**Leaf:** A two-item structure whose first item corresponds to the nibbles in the key not already accounted for by the accumulation of keys and branches traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be 1.

**Extension:** A two-item structure whose first item corresponds to a series of nibbles of size greater than one that are shared by at least two distinct keys past the accumulation of the keys of nibbles and the keys of branches as traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be 0.

**Branch:** A 17-item structure whose first sixteen items correspond to each of the sixteen possible nibble values for the keys at this point in their traversal. The 17th item is used in the case of this being a terminator node and thus a key being ended at this point in its traversal.

A branch is then only used when necessary; no branch nodes may exist that contain only a single non-zero entry. We may formally define this structure with the structural composition function *c*:

**** 0 ll 0ll − 1 ll ll ∃ ∈

HP(*I* [*i..*( *I* 1)]*,* 1)*, I* if I = 1 where *I* : *I* I

HP(*I*0[*i..*(*j* − 1)]*,* 0)*, n*(I*, j*) if *i* = *j* where *j* = max{*x* : ∃**l** : ll**l**ll = *x* ∧ ∀*I* ∈ I : *I*0[0*..*(*x* − 1)] = **l**}

****

(206) *c*(I*, i*) ≡ ****(*u*(0)*, u*(1)*, ..., u*(15)*, v*) otherwise where *u*(*j*) ≡ *n*f({*I* : *I* ∈ I ∧ *I*0[*i*] = *j*}*, i* + 1)

*v* =

****

*I*1 if ∃*I* : *I* ∈ I ∧ ll*I*0ll = *i*

() otherwise

D.1. **Trie Database.** Thus no explicit assumptions are made concerning what data is stored and what is not, since that is an implementation-specific consideration; we simply define the identity function mapping the key-value set I to a 32-byte hash and assert that only a single such hash exists for any I, which though not strictly true is accurate within acceptable precision given the Keccak hash’s collision resistance. In reality, a sensible implementation will not fully recompute the trie root hash for each set.

A reasonable implementation will maintain a database of nodes determined from the computation of various tries or, more formally, it will memoise the function *c*. This strategy uses the nature of the trie to both easily recall the contents of

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 22

any previous key-value set and to store multiple such sets in a very efficient manner. Due to the dependency relationship, Merkle-proofs may be constructed with an *O*(log *N* ) space requirement that can demonstrate a particular leaf must exist within a trie of a given root hash.

Appendix E. Precompiled Contracts

For each precompiled contract, we make use of a template function, ΞPRE, which implements the out-of-gas checking.

f≡

(207) Ξ

PRE

(***σ****, g, A, I*) (∅*,* 0*, A,* ()) if *g < g*r

(***σ****, g* − *g*r*, A,* **o**) otherwise

The precompiled contracts each use these definitions and provide specifications for the **o** (the output data) and *g*r, the gas requirements.

We define ΞECREC as a precompiled contract for the elliptic curve digital signature algorithm (ECDSA) public key recovery function (ecrecover). See Appendix [F](#_bookmark126) for the definition of the function ECDSARECOVER. We also define **d** to be the input data, well-defined for an infinite length by appending zeroes as required. In the case of an invalid signature (ECDSARECOVER(*h, v, r, s*) = ∅), we return no output.

(208)

(209)

(210)

ΞECREC ≡ ΞPRE where:

*g*r = 3000

fll ll

**o** = 0 if ECDSARECOVER(*h, v, r, s*) = ∅

32 otherwise

(211)

(212)

(213)

(214)

if ll**o**ll = 32 :

**o**[0*..*11] = 0

**o**[12*..*31] = KEC ECDSARECOVER(*h, v, r, s*) [12*..*31] where:

**d**[0*..*(ll[*I***d**](#_bookmark54)ll − 1)] = *I***d**

|  |  |  |  |
| --- | --- | --- | --- |
| (215) | **d**[ll*I***d**ll*..*] | = | (0*,* 0*, ...*) |
| (216) | *h* | = | **d**[0*..*31] |
| (217) | *v* | = | **d**[32*..*63] |
| (218) | *r* | = | **d**[64*..*95] |
| (219) | *s* | = | **d**[96*..*127] |

We define ΞSHA256 and ΞRIP160 as precompiled contracts implementing the SHA2-256 and RIPEMD-160 hash functions respectively. Their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words.

(220) ΞSHA256 ≡ ΞPRE where:

l

(221)

*g* = 60 + 12 ll*I***d**ll

32

r

(222) **o**[0*..*31] = SHA256(*I***d**)

(223) ΞRIP160 ≡ ΞPRE where:

l

(224)

*g* = 600 + 120 ll*I***d**ll

32

r

(225) **o**[0*..*11] = 0

(226) **o**[12*..*31] = RIPEMD160(*I***d**)

For the purposes here, we assume we have well-defined standard cryptographic functions for RIPEMD-160 and SHA2-256 of the form:

(227)

(228)

SHA256(**i** ∈ B) ≡ *o* ∈ B32

RIPEMD160(**i** ∈ B) ≡ *o* ∈ B20

The fourth contract, the identity function ΞID simply defines the output as the input:

(229) ΞID ≡ ΞPRE where:

l

(230)

*g* = 15 + 3 ll*I***d**ll

32

r

1. **o** = *I***d**

The fifth contract performs arbitrary-precision exponentiation under modulo. Here, 00 is taken to be one, and *x* mod 0 is zero for all *x*. The first word in the input specifies the number of bytes that the first non-negative integer *B* occupies. The second word in the input specifies the number of bytes that the second non-negative integer *E* occupies. The third word in the input specifies the number of bytes that the third non-negative integer *M* occupies. These three words are

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 23

followed by *B*, *E* and *M* . The rest of the input is discarded. Whenever the input is too short, the missing bytes are considered to be zero. The output is encoded big-endian into the same format as *M* ’s.

1. ΞEXPMOD ≡ ΞPRE except:

*G*

r

*g* = max 200*,* I *f*  max(*.€*M*, .€*B) max(*.€*E*,* 1)

*G*quaddivisor ≡ 3

*f* (*x*) ≡ *x* l2

8

****

quaddivisor

*.€*E =

0 if *.€*E ≤ 32 ∧ *E* = 0

llog2(*E*)J if *.€*E ≤ 32 ∧ *E*  = 0

8(*.€*E − 32) + llog2(*i*[(96 + *.€*B)*..*(127 + *.€*B)])J if 32 *< .€*E ∧ *i*[(96 + *.€*B)*..*(127 + *.€*B)] = 0

****

8(*.€*E − 32) otherwise

1. **o** = *BE* mod *M*  ∈ N8*£*

M

|  |  |  |  |
| --- | --- | --- | --- |
| (238) | *.€*B | ≡ | *i*[0*..*31] |
| (239) | *.€*E | ≡ | *i*[32*..*63] |
| (240) | *.€*M | ≡ | *i*[64*..*95] |
| (241) | *B* | ≡ | *i*[96*..*(95 + *.€*B)] |
| (242) | *E* | ≡ | *i*[(96 + *.€*B)*..*(95 + *.€*B + *.€*E)] |
| (243) | *M* | ≡ | *i*[(96 + *.€*B + *.€*E)*..*(95 + *.€*B + *.€*E + *.€*M)] |

(244)

*i*[*x*] *I***d**[*x*] if *x <* ll*I***d**ll

0 otherwise

f≡

* 1. **zkSNARK Related Precompiled Contracts.** We choose two numbers, both of which are prime.

(245) *p* ≡ 21888242871839275222246405745257275088696311157297823662689037894645226208583

(246) *q* ≡ 21888242871839275222246405745257275088548364400416034343698204186575808495617

Since *p* is a prime number, {0*,* 1*, . . . , p* − 1} forms a field with addition and multiplication modulo *p*. We call this field *F*p.

We define a set *C*1 with

(247) *C*1 ≡ {(*X, Y* ) ∈ *F*p × *F*p | *Y* 2 = *X*3 + 3} ∪ {(0*,* 0)}

We define a binary operation + on *C*1 for distinct elements (*X*1*, Y*1)*,* (*X*2*, Y*2) with

1

1

2

2

(0*,* 0) otherwise

(248)

(*X , Y* ) + (*X , Y* ) ≡ f(*X, Y* ) if *X*1 = *X*2

*λ Y*2 − *Y*1

≡

*X*2 − *X*1

*X* ≡ *λ*2 − *X*1 − *X*2

*Y* ≡ *λ*(*X*1 − *X*) − *Y*1

In the case where (*X*1*, Y*1) = (*X*2*, Y*2), we define + on *C*1 with

(249) (*X , Y* ) + (*X , Y* ) ≡ f(*X, Y* ) if *Y*1 0

1 1 2 2

*λ* ≡

(0*,* 0) otherwise

3*X*2

1

2*Y*1

*X* ≡ *λ*2 − 2*X*1

*Y* ≡ *λ*(*X*1 − *X*) − *Y*1

(*C*1*,* +) is known to form a group. We define scalar multiplication · with

(250) *n* · *P* ≡ (0*,* 0) + *P* + · · · + *P*

..,- .,

*n*

for a natural number *n* and a point *P* in *C*1.

We define *P*1 to be a point (1*,* 2) on *C*1. Let *G*1 be the subgroup of (*C*1*,* +) generated by *P*1. *G*1 is known to be a cyclic group of order *q*. For a point *P* in *G*1, we define log*P*1 (*P* ) to be the smallest natural number *n* satisfying *n* · *P*1 = *P* . log*P*1 (*P* ) is at most *q* − 1.

Let *Fp*2 be a field *Fp*[*i*]*/*(*i*2 + 1). We define a set *C*2 with

(251) *C*2 ≡ {(*X, Y* ) ∈ *Fp*2 × *Fp*2 | *Y* 2 = *X*3 + 3(*i* + 9)−1} ∪ {(0*,* 0)}

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 24

We define a binary operation + and scalar multiplication · with the same equations [(248),](#_bookmark121) [(249)](#_bookmark122) and [(250).](#_bookmark123) (*C*2*,* +) is also known to be a group. We define *P*2 in *C*2 with

(252)

*P*2 ≡ (11559732032986387107991004021392285783925812861821192530917403151452391805634 × *i*

+10857046999023057135944570762232829481370756359578518086990519993285655852781*,*

4082367875863433681332203403145435568316851327593401208105741076214120093531 × *i*

+8495653923123431417604973247489272438418190587263600148770280649306958101930)

We define *G*2 to be the subgroup of (*C*2*,* +) generated by *P*2. *G*2 is known to be the only cyclic group of order *q* on *C*2. For a point *P* in *G*2, we define log*P*2 (*P* ) be the smallest natural number *n* satisfying *n* · *P*2 = *P* . With this definition, log*P*2 (*P* ) is at most *q* − 1.

Let *GT* be the multiplicative abelian group underlying *Fq*12 . It is known that a non-degenerate bilinear map *e* : *G*1 × *G*2 → *GT* exists. This bilinear map is a type three pairing. There are several such bilinear maps, it does not matter which is chosen to be *e*. Let *PT* = *e*(*P*1*, P*2), *a* be a set of *k* points in *G*1, and *b* be a set of *k* points in *G*2. It follows from the definition of a pairing that the following are equivalent

(253)

(254)

log*P*1 (*a*1) × log*P*2 (*b*1) + · · · + log*P*1 (*ak*) × log*P*2 (*bk*) ≡ 1 mod *q*

*k*

*e* (*ai, bi*) = *PT*

*i*=0

Thus the pairing operation provides a method to verify [(253).](#_bookmark124)

A 32 byte number **x** ∈ **P**256 might and might not represent an element of *F*p.

(255) *δ* (**x**) ≡ f**x** if **x** *< p*

p

∅ otherwise

A 64 byte data **x** ∈ **B**512 might and might not represent an element of *G*1.

(256)

*δ* (**x**) ≡ f*g*1 if *g*1 ∈ *G*1

(257)

1

∅ otherwise

1. (*x, y*) if *x* = ∅ ∧ *y* ∅

1

∅ otherwise

(258) *x* ≡ *δ*p(**x**[0*..*31])

f≡

(259) *y* ≡ *δ*p(**x**[32*..*63])

A 128 byte data **x** ∈ **B**1024 might and might not represent an element of *G*2.

(260)

f≡

*δ* (**x**) ≡ f*g*2 if *g*2 ∈ *G*2

(261)

∅ otherwise

*g* ((*x*0*i* + *y*0)*,* (*x*1*i* + *y*1)) if *x*0 = ∅ ∧ *y*0 = ∅ ∧ *x*1

2

∅ otherwise

∅ ∧ *y*1 = ∅

2

|  |  |  |  |
| --- | --- | --- | --- |
| (262) | *x*0 | ≡ | *δ*p(**x**[0*..*31]) |
| (263) | *y*0 | ≡ | *δ*p(**x**[32*..*63]) |
| (264) | *x*1 | ≡ | *δ*p(**x**[64*..*95]) |
| (265) | *y*1 | ≡ | *δ*p(**x**[96*..*127]) |

We define ΞSNARKV as a precompiled contract which checks if [(253)](#_bookmark124) holds, for intended use in zkSNARK verification.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 25

1. ΞSNARKV ≡ ΞPRE except:
2. ΞSNARKV(***σ****, g, A, I*) = (∅*,* 0*, A,* ()) if *F*

*F* ≡ (ll*I***d**ll mod 192 = 0 ∨ (∃*j. a*j = ∅ ∨ *b*j = ∅))

*k* = ll*I***d**ll

192

*g*r = 34000*k* + 45000

f≡

**o**[0*..*31] 0*x*0000000000000000000000000000000000000000000000000000000000000001 if *v* ∧ ¬*F*

0*x*0000000000000000000000000000000000000000000000000000000000000000 if ¬*v* ∧ ¬*F*

*v* ≡ (log*P*1 (*a*1) × log*P*2 (*b*1) + · · · + log*P*1 (*ak*) × log*P*2 (*bk*) ≡ 1 mod *q*) *a*1 ≡ *δ*1(*I***d**[0*..*63])

*b*1 ≡ *δ*2(*I***d**[64*..*191])

.

*ak* ≡ *δ*1(*I***d**[(ll*I***d**ll − 192)*..*(ll*I***d**ll − 129)])

*bk* ≡ *δ*2(*I***d**[(ll*I***d**ll − 128)*..*(ll*I***d**ll − 1)])

We define a precompiled contract for addition on *G*1.

1. (277)

ΞBN ADD ≡ ΞBN PRE except:

ΞBN ADD(***σ****, g, A, I*) = (∅*,* 0*, A,* ()) if *x* = ∅ ∨ *y* = ∅

*g*r = 150

**o** ≡ *δ*−1(*x* + *y*) where + is the group operation in *G*1 *x δ*1 *I*¯**d**[0*..*63]

≡

1

*y* ≡ *δ*1 *I*¯**d**[64*..*127]

f*I* [*x*] if *x <* ll*I* ll**d d**¯*I* [*x*] ≡

**d**

0 otherwise

We define a precompiled contract for scalar multiplication on *G*1, where *I*¯**d** is defined in [(283).](#_bookmark125)

ΞBN MUL ≡ ΞPRE except:

ΞBN MUL(***σ****, g, A, I*) = (∅*,* 0*, A,* ()) if *x* = ∅

*g*r = 6000

**o** ≡ *δ*−1(*n* · *x*) where · is the scalar multiplication in *G*1 *x* ≡ *δ*1 *I*¯**d**[0*..*63]

1

(289) *n* ≡ *I*¯**d**[64*..*95]

* 1. **BLAKE2 Precompiled Contract.** EIP-152 by [Hess et al.](#_bookmark104) [[2016]](#_bookmark104) defines ΞBLAKE2 F as a precompiled contract implementing the compression function F used in the BLAKE2 cryptographic hashing algorithm. The F compression function is specified in RFC 7693 by [Saarinen and Aumasson [2015].](#_bookmark92)

(290)

(291)

(292)

(293)

ΞBLAKE2 F ≡ ΞPRE except:

ΞBLAKE2 F(***σ****, g, A, I*) = (∅*,* 0*, A,* ()) if ll*I***d**ll =

*g*r = *r*

**o** ≡ LE8(*h*0) · *...* · LE8(*h*7)

213 ∨ *f* ∈*/* {0*,* 1}

|  |  |  |  |
| --- | --- | --- | --- |
| (294) | (*h*0*, . . . , h*7) | ≡ | F(*h, m, t*low*, t*high*, f* ) with *r* rounds and *w* = 64 |
| (295) | BE4(*r*) | ≡ | *I***d**[0*..*4] |
| (296) | LE8(*h*0) | ≡ | *I***d**[4*..*12] |
| (297) |  | *. . .* |  |
| (298) | LE8(*h*7) | ≡ | *I***d**[60*..*68] |
| (299) | LE8(*m*0) | ≡ | *I***d**[68*..*76] |
| (300) |  | *. . .* |  |
| (301) | LE8(*m*15) | ≡ | *I***d**[188*..*196] |
| (302) | LE8(*t*low) | ≡ | *I***d**[196*..*204] |
| (303) | LE8(*t*high) | ≡ | *I***d**[204*..*212] |
| (304) | *f* | ≡ | *I***d**[212] |

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 26

where *r* ∈ B32, ∀*i* ∈ 0*..*7 : *hi* ∈ B64, ∀*i* ∈ 0*..*15 : *mi* ∈ B64, *t*low ∈ B64, *t*high ∈ B64, *f* ∈ B8, BE*k* is the *k*-byte big-endian representation—compare [with(193):](#_bookmark115)

*k*−1

*k* 1 *n*

(305) BE*k*(*x*) ≡ (*b*0*, b*1*, ..., bk*−1) : *x* = *bn* · 256 − −

*n*=0

and LE*k* is the *k*-byte little-endian representation:

*k*−1

(306) LE*k*(*x*) ≡ (*b*0*, b*1*, ..., bk*−1) : *x* = *bn* · 256*n*

*n*=0

Appendix F. Signing Transactions

Transactions are signed using recoverable ECDSA signatures. This method utilises the SECP-256k1 curve as described by [Courtois et al. [2014],](#_bookmark84) and is implemented similarly to as described by [Gura et al.](#_bookmark99) [[2004]](#_bookmark99) on p. 9 of 15, para. 3.

It is assumed that the sender has a valid private key *p*r, which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range [1*,* secp256k1n − 1].

We assume the existence of functions ECDSAPUBKEY, ECDSASIGN and ECDSARECOVER. These are formally defined in the

literature, e.g. by [Johnson et al. [2001].](#_bookmark108)

(307)

(308)

(309)

ECDSAPUBKEY(*p*r ∈ B32) ≡ *p*u ∈ B64

ECDSASIGN(*e* ∈ B32*, p*r ∈ B32) ≡ (*v* ∈ B1*, r* ∈ B32*, s* ∈ B32)

ECDSARECOVER(*e* ∈ B32*, v* ∈ B1*, r* ∈ B32*, s* ∈ B32) ≡ *p*u ∈ B64

Where *p*u is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each *<* 2256), *p*r is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range) and *e* is the hash of the transaction, [*h*(*T*](#_bookmark130) ). It is assumed that *v* is the ‘recovery identifier’. The recovery identifier

is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which *r* is the x-value; this value is in the range of [0*,* 3], however we declare the upper two possibilities, representing infinite values, invalid. The value 0 represents an even *y* value and 1 represents an odd *y* value.

We declare that an ECDSA signature is invalid unless all the following conditions are true:

(310)

(311)

(312)

where: (313)

0 *<**r <* secp256k1n

0 *< s <* secp256k1n ÷ 2 + 1

[*v*](#_bookmark127) ∈ {0*,* 1}

secp256k1n = 115792089237316195423570985008687907852837564279074904382605163141518161494337

For a given private key, *p*r, the Ethereum address *A*(*p*r) (a 160-bit value) to which it corresponds is defined as the rightmost 160-bits of the Keccak-256 hash of the corresponding ECDSA public key:

(314) *A*(*p*r) = B96*..*255 KEC ECDSAPUBKEY(*p*r)

The message hash, *h*(*T* ), to be signed is the Keccak-256 hash of the transaction. Three different flavours of signing schemes are available:



(315)

*L*X(*T* ) ≡

where

(*T*n*, T*p*, T*g*, T*t*, T*v*,* **p**) if *T*x = 0 ∧ *T*w ∈ {27*,* 28}

(*T*n*, T*p*, T*g*, T*t*, T*v*,* **p***, β,* ()*,* ()) if *T*x = 0 ∧ *T*w ∈ {2*β* + 35*,* 2*β* + 36}



(*T*c*, T*n*, T*p*, T*g*, T*t*, T*v*,* **p***, T***A**) if *T*x = 1

f≡

(316)

**p** *T***i** if *T*t = ∅

*T***d** otherwise

f≡

*h*(*T* ) KEC(RLP(*L*X(*T* ))) if *T*x = 0

KEC(*T*x · RLP(*L*X(*T* ))) otherwise

The signed transaction *G*(*T, p*r) is defined as: (317)

*G*(*T, p*r) ≡ *T* except:

(318)

[Reiterating from previously:](#_bookmark8) (319)

(320)

(*T*y*, T*r*, T*s) = ECDSASIGN(*h*(*T* )*, p*r)

*T*r = [*r*](#_bookmark128) *T*s = [*s*](#_bookmark129)

and [*T*w](#_bookmark11) of legacy transcations is either 27 + *T*y or 2[*β*](#_bookmark1) + 35 + *T*y.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 27

|  |  |
| --- | --- |
| We may then define the sender function *S* of the transaction as: | |
| (321) | *S*(*T* ) ≡ B96*..*255 KEC ECDSARECOVER(*h*(*T* )*, v, T*r*, T*s) |
| (322) | *T*w − 27 if *T*x = 0 ∧ *T*w ∈ {27*,* 28}  *v* ≡ (*T*w − 35) mod 2 if *T*x = 0 ∧ *T*w ∈ {2*β* + 35*,* 2*β* + 36}  *T*y if *T*x = 1 |
| The assertion that the sender of a signed transaction equals the address of the signer should be self-evident:  (323) ∀*T* : ∀*p*r : *S*(*G*(*T, p*r)) ≡ *A*(*p*r)  Appendix G. Fee Schedule  The fee schedule *G* is a tuple of scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect. | |

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| *G*zero | 0 | Nothing paid for operations of the set *W*zero. |
| *G*jumpdest | 1 | Amount of gas to pay for a JUMPDEST operation. |
| *G*base | 2 | Amount of gas to pay for operations of the set *W*base. |
| *G*verylow | 3 | Amount of gas to pay for operations of the set *W*verylow. |
| *G*low | 5 | Amount of gas to pay for operations of the set *W*low. |
| *G*mid | 8 | Amount of gas to pay for operations of the set *W*mid. |
| *G*high | 10 | Amount of gas to pay for operations of the set *W*high. |
| *G*warmaccess | 100 | Cost of a warm account or storage access. |
| *G*accesslistaddress | 2400 | Cost of warming up an account with the access list. |
| *G*accessliststorage | 1900 | Cost of warming up a storage with the access list. |
| *G*coldaccountaccess | 2600 | Cost of a cold account access. |
| *G*coldsload | 2100 | Cost of a cold storage access. |
| *G*sset | 20000 | Paid for an SSTORE operation when the storage value is set to non-zero from zero. |
| *G*sreset | 2900 | Paid for an SSTORE operation when the storage value’s zeroness remains unchanged or is set to zero. |
| *R*sclear | 15000 | Refund given (added into refund counter) when the storage value is set to zero from  non-zero. |
| *R*selfdestruct | 24000 | Refund given (added into refund counter) for self-destructing an account. |
| *G*selfdestruct | 5000 | Amount of gas to pay for a SELFDESTRUCT operation. |
| *G*create | 32000 | Paid for a CREATE operation. |
| *G*codedeposit | 200 | Paid per byte for a CREATE operation to succeed in placing code into state. |
| *G*callvalue | 9000 | Paid for a non-zero value transfer as part of the CALL operation. |
| *G*callstipend | 2300 | A stipend for the called contract subtracted from *G*callvalue for a non-zero value transfer. |
| *G*newaccount | 25000 | Paid for a CALL or SELFDESTRUCT operation which creates an account. |
| *G*exp | 10 | Partial payment for an EXP operation. |
| *G*expbyte | 50 | Partial payment when multiplied by the number of bytes in the exponent for the EXP operation. |
| *G*memory | 3 | Paid for every additional word when expanding memory. |
| *G*txcreate | 32000 | Paid by all contract-creating transactions after the *Homestead* transition. |
| *G*txdatazero | 4 | Paid for every zero byte of data or code for a transaction. |
| *G*txdatanonzero | 16 | Paid for every non-zero byte of data or code for a transaction. |
| *G*transaction | 21000 | Paid for every transaction. |
| *G*log | 375 | Partial payment for a LOG operation. |
| *G*logdata | 8 | Paid for each byte in a LOG operation’s data. |
| *G*logtopic | 375 | Paid for each topic of a LOG operation. |
| *G*keccak256 | 30 | Paid for each KECCAK256 operation. |
| *G*keccak256word | 6 | Paid for each word (rounded up) for input data to a KECCAK256 operation. |
| *G*copy | 3 | Partial payment for \*COPY operations, multiplied by words copied, rounded up. |
| *G*blockhash | 20 | Payment for each BLOCKHASH operation. |

Appendix H. Virtual Machine Specification

When interpreting 256-bit binary values as integers, the representation is big-endian.

When a 256-bit machine datum is converted to and from a 160-bit address or hash, the rightwards (low-order for BE) 20 bytes are used and the leftmost 12 are discarded or filled with zeroes, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 28

* 1. **Gas Cost.** The general gas cost function, *C*, is defined as:

(324)

*C*(***σ****,* ***µ****, A, I*) ≡ *C*mem(***µ***i) − *C*mem(***µ***i) +

****

*C*SSTORE(***σ****,* ***µ****, A, I*) if *w* = SSTORE

*G*exp if *w* = EXP ∧ ***µ*s**[1] = 0

****

****

*G*exp + *G*expbyte × (1 + llog256(***µ*s**[1])J) if *w* = EXP ∧ ***µ*s**[1] *>* 0 *G*verylow + *G*copy × r***µ*s**[2] ÷ 321 if *w* ∈ *W*copy *C*aaccess(***µ*s**[0] mod 2160*, A*) + *G*copy × r***µ*s**[3] ÷ 321 if *w* = EXTCODECOPY *C*aaccess(***µ*s**[0] mod 2160*, A*) if *w* ∈ *W*extaccount

****

****

****

*G*log + *G*logdata × ***µ*s**[1] if *w* = LOG0

****

*G*log + *G*logdata × ***µ*s**[1] + *G*logtopic if *w* = LOG1

*G*log + *G*logdata × ***µ*s**[1] + 2*G*logtopic if *w* = LOG2

****

*G*log + *G*logdata × ***µ*s**[1] + 3*G*logtopic if *w* = LOG3

*G*log + *G*logdata × ***µ*s**[1] + 4*G*logtopic if *w* = LOG4

****

*C*CALL(***σ****,* ***µ****, A*) if *w* ∈ *W*call

*C*SELFDESTRUCT(***σ****,* ***µ***) if *w* = SELFDESTRUCT

*G*create if *w* = CREATE

****

*G*create + *G*keccak256word × r***µ*s**[2] ÷ 321 if *w* = [CREATE2](#_bookmark150) *G*keccak256 + *G*keccak256word × r***µ*s**[1] ÷ 321 if *w* = KECCAK256 *G*jumpdest if *w* = JUMPDEST

****

*C*SLOAD(***µ****, A, I*) if *w* = SLOAD

****

*G*zero if *w* ∈ *W*zero

*G*base if *w* ∈ *W*base

****

*G*verylow if *w* ∈ *W*verylow

*G*low if *w* ∈ *W*low

****

*G*mid if *w* ∈ *W*mid

*G*high if *w* ∈ *W*high

*G*blockhash if *w* = [BLOCKHASH](#_bookmark144)

f≡

1. *w I***b**[***µ***pc] if ***µ***pc *<* ll*I***b**ll

STOP otherwise

where:

1. *C*mem(*a*) ≡ *G*memory · *a* +

*a*2

512

1. *C*

aaccess

(*x, A*) *G*warmaccess if *x* ∈ [*A***a**](#_bookmark39)*G*coldaccountaccess otherwise

with *C*CALL, *C*SELFDESTRUCT, *C*SLOAD and *C*SSTORE as specified in the appropriate section below. We define the following subsets of instructions:

f≡

*W*zero = {STOP, RETURN, REVERT}

*W*base = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS}

*W*verylow = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH\*, DUP\*, SWAP\*}

*W*low = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND, SELFBALANCE}

*W*mid = {ADDMOD, MULMOD, JUMP}

*W*high = {JUMPI}

*W*copy = {CALLDATACOPY, CODECOPY, RETURNDATACOPY} *W*call = {CALL, CALLCODE, DELEGATECALL, STATICCALL} *W*extaccount = {BALANCE, EXTCODESIZE, EXTCODEHASH}

Note the memory cost component, given as the product of *G*memory and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, ***µ***i in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes.

Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory to be extended to the beginning of the range. ***µ***i is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 29

Note also that *C*mem is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

While defining the instruction set, we defined the memory-expansion for range function, *M* , thus:

1. *M* (*s, f, l*) *s* if *l* = 0 max(*s,* r(*f* + *l*) ÷ 321) otherwise

f≡

Another useful function is “all but one 64th” function *L* defined as:

(329) *L*(*n*) ≡ *n* − l*n/*64J

* 1. **Instruction Set.** As previously specified in section [9,](#_bookmark52) these definitions take place in the final context there. In particular we assume *O* is the EVM state-progression function and define the terms pertaining to the next cycle’s state (***σ*** *,* ***µ*** ) such that:

(330) *O*(***σ****,* ***µ****, A, I*) ≡ (***σ*** *,* ***µ*** *, A , I*) with exceptions, as noted

Here given are the various exceptions to the state transition rules given in section [9](#_bookmark52) specified for each instruction, together with the additional instruction-specific definitions of *J* and *C*. For each instruction, also specified is *α*, the additional items placed on the stack and *δ*, the items removed from stack, as defined in section [9.](#_bookmark52)

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 30

# 0s: Stop and Arithmetic Operations

All arithmetic is modulo 2256 unless otherwise noted. The zero-th power of zero 00 is defined to be one.

# Value Mnemonic *δ α* Description

0x00 STOP 0 0 Halts execution.

0x01 ADD 2 1 Addition operation.

***µ*s**[0] ≡ ***µ*s**[0] + ***µ*s**[1]

0x02 MUL 2 1 Multiplication operation.

***µ*s**[0] ≡ ***µ*s**[0] × ***µ*s**[1]

0x03 SUB 2 1 Subtraction operation.

***µ*s**[0] ≡ ***µ*s**[0] − ***µ*s**[1]

0x04 DIV 2 1 Integer division operation.

0 if ***µ*s**[1] = 0

f***µ*** [0] ≡

**s**

l***µ*s**[0] ÷ ***µ*s**[1]J otherwise



0x05 SDIV 2 1 Signed integer division operation (truncated).

255 255

**sgn**(***µ*s**[0] ÷ ***µ*s**[1])l|***µ*s**[0] ÷ ***µ*s**[1]|J otherwise

Where all values are treated as two’s complement signed 256-bit integers. Note the overflow semantic when −2255 is negated.

**s**

***µ*s**[0] ≡

−2 if ***µ*** [0] = −2 ∧ ***µ*** [1] = −1

0 if ***µ*s**[1] = 0

**s**

0x06 MOD 2 1 Modulo remainder operation.

***µ*** [0] ≡ 0 if ***µ*s**[1] = 0

**s**

**s**

f

**s**

***µ*** [0] mod ***µ*** [1] otherwise

0x07 SMOD 2 1 Signed modulo remainder operation.

f

***µ*** [0] ≡ 0 if ***µ*s**[1] = 0

**s**

**sgn**(***µ*** [0])(|***µ*** [0]| mod |***µ*** [1]|) otherwise

**s**

**s**

**s**

Where all values are treated as two’s complement signed 256-bit integers.

0x08 ADDMOD 3 1 Modulo addition operation.

***µ*** [0] ≡ 0 if ***µ*s**[2] = 0

**s**

**s**

**s**

f

**s**

(***µ*** [0] + ***µ*** [1]) mod ***µ*** [2] otherwise

All intermediate calculations of this operation are not subject to the 2256 modulo.

0x09 MULMOD 3 1 Modulo multiplication operation.

***µ*** [0] ≡ 0 if ***µ*s**[2] = 0

f

**s**

(***µ*** [0] × ***µ*** [1]) mod ***µ*** [2] otherwise

**s**

**s**

**s**

All intermediate calculations of this operation are not subject to the 2256 modulo.

0x0a EXP 2 1 Exponential operation.

***µ*** [0] ≡ ***µ*** [0]***µ*s**[1]

**s**

**s**

0x0b SIGNEXTEND 2 1 Extend length of two’s complement signed integer.

∀*i* ∈ [0*..*255] : ***µ*** [0]

**s**

≡ f***µ*s**[1]t if *i*  *t* where *t* = 256 − 8(***µ*s**[0] + 1)

***µ*s**[*x*]i gives the *i*th bit (counting from zero) of ***µ*s**[*x*]

i

***µ*s**[1]

i

otherwise

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 31

# 10s: Comparison & Bitwise Logic Operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0x10 | LT | 2 | 1 | Less-than comparison. |

***µ*** [0] ≡ f1 if ***µ*s**[0] *<* ***µ*s**[1]

**s**

0 otherwise

0x11 GT 2 1 Greater-than comparison.

f

***µ*** [0] ≡ 1 if ***µ*s**[0] *>* ***µ*s**[1]

**s**

0 otherwise

0x12 SLT 2 1 Signed less-than comparison.

f

***µ*** [0] ≡ 1 if ***µ*s**[0] *<* ***µ*s**[1]

**s**

0 otherwise

Where all values are treated as two’s complement signed 256-bit integers.

0x13 SGT 2 1 Signed greater-than comparison.

***µ*** [0] ≡ 1 if ***µ*s**[0] *>* ***µ*s**[1]

f

**s**

0 otherwise

Where all values are treated as two’s complement signed 256-bit integers.

0x14 EQ 2 1 Equality comparison.

***µ*** [0] ≡ 1 if ***µ*s**[0] = ***µ*s**[1]

f

**s**

0 otherwise

0x15 ISZERO 1 1 Simple not operator.

***µ*** [0] ≡ 1 if ***µ*s**[0] = 0

f

**s**

0 otherwise

0x16 AND 2 1 Bitwise AND operation.

∀*i* ∈ [0*..*255] : ***µ*s**[0]i ≡ ***µ*s**[0]i ∧ ***µ*s**[1]i

0x17 OR 2 1 Bitwise OR operation.

∀*i* ∈ [0*..*255] : ***µ*s**[0]i ≡ ***µ*s**[0]i ∨ ***µ*s**[1]i

0x18 XOR 2 1 Bitwise XOR operation.

∀*i* ∈ [0*..*255] : ***µ*s**[0]i ≡ ***µ*s**[0]i ⊕ ***µ*s**[1]i

f≡

0x19 NOT 1 1 Bitwise NOT operation.

∀*i* ∈ [0*..*255] : ***µ*s**[0]i

1 if ***µ*s**[0]i = 0

0 otherwise

0x1a BYTE 2 1 Retrieve single byte from word.

f≡

∀*i* ∈ [0*..*255] : ***µ*s**[0]i

***µ*s**[1](*i*−248+8***µ*s**[0]) if *i* ≥ 248 ∧ ***µ*s**[0] *<* 32

0 otherwise

For the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

|  |  |  |  |
| --- | --- | --- | --- |
| 0x1b | SHL | 2 | 1 Left shift operation.  ***µ*** [0] ≡ (***µ*** [1] × 2***µ*s**[0]) mod 2256  **s s** |
| 0x1c | SHR | 2 | 1 Logical right shift operation.  ***µ*** [0] ≡ l***µ*** [1] ÷ 2***µ*s**[0]J  **s s** |
| 0x1d | SAR | 2 | 1 Arithmetic (signed) right shift operation. |

***µ*** [0] ≡ l***µ*** [1] ÷ 2***µ*s**[0]J

**s**

**s**

Where ***µ*s**[0] and ***µ*s**[1] are treated as two’s complement signed 256-bit integers, while ***µ*s**[0] is treated as unsigned.

# 20s: KECCAK256

|  |  |  |  |
| --- | --- | --- | --- |
| **Value** | **Mnemonic** *δ* | *α* | **Description** |
| 0x20 | KECCAK256 2 | 1 | Compute Keccak-256 hash.  ***µ*s**[0] ≡ KEC(***µ*m**[***µ*s**[0] *. . .* (***µ*s**[0] + ***µ*s**[1] − 1)])  ***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[1]) |

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 32

∀*i* ∈ {0 *. . .* ***µ*** [2] − 1} : ***µ***

**s**

[***µ*** [0] + *i*] ≡ f*I***d**[***µ*s**[1] + *i*] if ***µ*s**[1] + *i <* ll*I***d**ll

The additions in ***µ*** [1] + *i* are not subject to the 2256 modulo.

**s**

**m**

**s**

0 otherwise

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | **30s: Environmental Information** |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0x30 | ADDRESS | 0 | 1 | Get address of currently executing account. |
| 0x31 | BALANCE | 1 | 1 | ***µ*s**[0] ≡ *I*a  Get balance of the given account. |
|  |  |  |  | f***σ***[***µ*** [0] mod 2160]b if ***σ***[***µ*** [0] mod 2160] = ∅  ***µ*** [0] ≡ **s s**  **s** 0 otherwise  *A***a** ≡ *A***a** ∪ {***µ*s**[0] mod 2 }  160 |
| 0x32 | ORIGIN | 0 | 1 | Get execution origination address.  ***µ*s**[0] ≡ *I*o  This is the sender of original transaction; it is never an account with  non-empty associated code. |
| 0x33 | CALLER | 0 | 1 | Get caller address.  ***µ*s**[0] ≡ *I*s  This is the address of the account that is directly responsible for  this execution. |
| 0x34 | CALLVALUE | 0 | 1 | Get deposited value by the instruction/transaction responsible for this execution.  ***µ*s**[0] ≡ *I*v |
| 0x35 | CALLDATALOAD | 1 | 1 | Get input data of current environment.  ***µ*s**[0] ≡ *I***d**[***µ*s**[0] *. . .* (***µ*s**[0] + 31)] with *I***d**[*x*] = 0 if *x* ;;: ll*I***d**ll This pertains to the input data passed with the message call instruction or transaction. |
| 0x36 | CALLDATASIZE | 0 | 1 | Get size of input data in current environment.  ***µ*s**[0] ≡ ll*I***d**ll  This pertains to the input data passed with the message call  instruction or transaction. |
| 0x37 | CALLDATACOPY | 3 | 0 | Copy input data in current environment to memory. |

***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[2])

This pertains to the input data passed with the message call instruction

or transaction.

0x38 CODESIZE 0 1 Get size of code running in current environment.

***µ*s**[0] ≡ ll*I***b**ll

0x39 CODECOPY 3 0 Copy code running in current environment to memory.

∀*i* ∈ {0 *. . .* ***µ*** [2] − 1} : ***µ***

[***µ*** [0] + *i*] ≡ f*I***b**[***µ*s**[1] + *i*] if ***µ*s**[1] + *i <* ll*I***b**ll

**s m s**

***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[2])

STOP otherwise

The additions in ***µ*** [1] + *i* are not subject to the 2256 modulo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | **s** |
| 0x3a | GASPRICE | 0 | 1 | Get price of gas in current environment.  ***µ*s**[0] ≡ *I*p  This is gas price specified by the originating transaction. |
| 0x3b | EXTCODESIZE | 1 | 1 | Get size of an account’s code. |

***µ*s**[0] ≡

160

**s**

fll**b**ll if ***σ***[***µ*** [0] mod 2 ] = ∅

0 otherwise

where KEC(**b**) ≡ ***σ***[***µ*** [0] mod 2160]c

**s**

*A***a** ≡ *A***a** ∪ {***µ*** [0] mod 2160}

**s**

0x3c EXTCODECOPY 4 0 Copy an account’s code to memory.

∀*i* ∈ {0 *. . .* ***µ*** [3] − 1} : ***µ***

**s**

[***µ*** [1] + *i*] ≡ f**b**[***µ*s**[2] + *i*] if ***µ*s**[2] + *i <* ll**b**ll

where KEC(**b**) ≡ ***σ***[***µ*** [0] mod 2160]c

**s**

**m**

**s**

STOP otherwise

We assume **b** ≡ () if ***σ***[***µ*** [0] mod 2160] = ∅.

**s**

***µ***i *M* (***µ***i*,* ***µ*s**[1]*,* ***µ*s**[3])

≡

The additions in ***µ*** [2] + *i* are not subject to the 2256 modulo.

**s**

*A***a** ≡ *A***a** ∪ {***µ*** [0] mod 2160}

**s**

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 33

0x3d RETURNDATASIZE 0 1 Get size of output data from the previous call from the current

environment.

***µ*s**[0] ≡ ll***µ*o**ll

0x3e RETURNDATACOPY 3 0 Copy output data from the previous call to memory.

∀*i* ∈ {0 *. . .* ***µ*** [2] − 1} : ***µ***

**s**

[***µ*** [0] + *i*] ≡ f***µ*o**[***µ*s**[1] + *i*] if ***µ*s**[1] + *i <* ll***µ*o**ll

The additions in ***µ*** [1] + *i* are not subject to the 2256 modulo.

**s**

**m**

**s**

0 otherwise

***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[2])

0x3f EXTCODEHASH 1 1 Get hash of an account’s code.

0 if DEAD(***σ****,* ***µ*** [0] mod 2160)

***σ***[***µ*** [0] mod 2160]c otherwise

f

***µ*s**[0] ≡

**s**

*A***a** ≡ *A***a** ∪ {***µ*** [0] mod 2160}

# 40s: Block Information

**s**

**s**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0x40 | BLOCKHASH | 1 | 1 | Get the hash of one of the 256 most recent complete blocks.  ***µ*s**[0] ≡ *P* (*I*Hp *,* ***µ*s**[0]*,* 0) |

where *P* is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than or

equal to the current block number or more than 256 blocks behind the current block.



*P* (*h, n, a*) ≡

1. if *n* = *H*i

*P* (*H*p*, n, a* + 1) otherwise

0 if *n > H*i ∨ *a* = 256 ∨ *h* = 0

and we assert the header *H* can be determined from its hash *h* unless *h* is zero (as is the case for the parent hash of the genesis block).

0x41 COINBASE 0 1 Get the current block’s beneficiary address.

***µ*s**[0] ≡ *I*Hc

0x42 TIMESTAMP 0 1 Get the current block’s timestamp.

***µ*s**[0] ≡ *I*Hs

0x43 NUMBER 0 1 Get the current block’s number.

***µ*s**[0] ≡ *I*Hi

0x44 DIFFICULTY 0 1 Get the current block’s difficulty.

***µ*s**[0] ≡ *I*Hd

0x45 GASLIMIT 0 1 Get the current block’s gas limit.

***µ*s**[0] ≡ *I*Hl

0x46 CHAINID 0 1 Get the [chain ID.](#_bookmark1)

***µ*s**[0] ≡ *β*

0x47 SELFBALANCE 0 1 Get balance of currently executing account.

***µ*****s**[0] ≡ ***σ***[*I*a]b

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 34

# 50s: Stack, Memory, Storage and Flow Operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value**  0x50 | **Mnemonic**  POP | *δ*  1 | *α*  0 | **Description**  Remove item from stack. |
| 0x51 | MLOAD | 1 | 1 | Load word from memory.  ***µ*s**[0] ≡ ***µ*m**[***µ*s**[0] *. . .* (***µ*s**[0] + 31)]  ***µ***i ≡ max(***µ***i*,* r(***µ*s**[0] + 32) ÷ 321)  The addition in the calculation of ***µ*** is not subject to the 2256 modulo.  i |
| 0x52 | MSTORE | 2 | 0 | Save word to memory.  ***µ*m**[***µ*s**[0] *. . .* (***µ*s**[0] + 31)] ≡ ***µ*s**[1] ***µ***i ≡ max(***µ***i*,* r(***µ*s**[0] + 32) ÷ 321)  The addition in the calculation of ***µ*** is not subject to the 2256 modulo.  i |
| 0x53 | MSTORE8 | 2 | 0 | Save byte to memory.  ***µ*m**[***µ*s**[0]] ≡ (***µ*s**[1] mod 256)  ***µ***i ≡ max(***µ***i*,* r(***µ*s**[0] + 1) ÷ 321)  The addition in the calculation of ***µ*** is not subject to the 2256 modulo.  i |
| 0x54 | SLOAD | 1 | 1 | Load word from storage. |

***µ*s**[0] ≡ ***σ***[*I*a]**s**[***µ*s**[0]]

*A***K** ≡ *A***K** ∪ {(*I*a*,* ***µ***f**s**[0])}

*C*

SLOAD

(***µ****, A, I*) ≡

*G*warmaccess if (*I*a*,* ***µ*s**[0]) ∈ *A***K**

*G*coldsload otherwise

0x55 SSTORE 2 0 Save word to storage.

***σ***[*I*a]**s**[***µ*s**[0]] ≡ ***µ*s**[1]

*A***K** ≡ *A***K** ∪ {(*I*a*,* ***µ*s**[0])}

*C*SSTORE(***σ****,* ***µ***) and *A*r are specified by EIP-2200 as follows.

We remind the reader that the checkpoint (“original”) state [***σ***0](#_bookmark41) is the state if the current transaction were to revert.

Let *v*0 = ***σ***0[*I*a]**s**[***µ*s**[0]] be the original value of the storage slot. Let *v* = ***σ***[*I*a]**s**[***µ*s**[0]] be the current value.

Let *v* = ***µ*s**[1] be the new value.

Then:

f≡

*C*SSTORE

(***σ****,* ***µ****, A, I*) 0 if (*I*a*,* ***µ*s**[0]) ∈ *A***K** *G*coldsload otherwise

*G*warmaccess if *v* = *v* ∨ *v*0 = *v*

+ *G*sset if *v* = *v* ∧ *v*0 = *v* ∧ *v*0 = 0



 *G*sreset if *v* = *v* ∧ *v*0 = *v* ∧ *v*0 = 0

*R*sclear if *v*  = *v* ∧ *v*0 = *v* ∧ *v* = 0

*A*r ≡ *A*r +

*r*dirtyclear + *r*dirtyreset if *v*

*v* ∧ *v*0 = *v*

where

0 otherwise

−*R*sclear if *v*0 = 0 ∧ *v* = 0

*r*dirtyclear ≡



*r*dirtyreset ≡

*R*sclear if *v*0 = 0 ∧ *v* = 0

0 otherwise

*G*sset − *G*warmaccess if *v*0 = *v* ∧ *v*0 = 0



*G*sreset − *G*warmaccess if *v*0 = *v* ∧ *v*0 = 0



0 otherwise

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0x56 | JUMP | 1 | 0 Alter the program counter. | |
|  |  |  |  | *J*JUMP(***µ***) ≡ ***µ*s**[0]  This has the effect of writing said value to ***µ***pc. See section [9.](#_bookmark52) |
| 0x57 | JUMPI | 2 | 0 | Conditionally alter the program counter. |

*J*JUMPI

(***µ***) ***µ*s**[0] if ***µ*s**[1] = 0

***µ***pc + 1 otherwise

f≡

This has the effect of writing said value to ***µ***pc. See section [9.](#_bookmark52)

0x58 PC 0 1 Get the value of the program counter *prior* to the increment corresponding to this instruction.

***µ*s**[0] ≡ ***µ***pc

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 35

0x59 MSIZE 0 1 Get the size of active memory in bytes.

***µ*s**[0] ≡ 32***µ***i

0x5a GAS 0 1 Get the amount of available gas, including the corresponding reduction for the cost of this instruction.

***µ*s**[0] ≡ ***µ***g

0x5b JUMPDEST 0 0 Mark a valid destination for jumps.

This operation has no effect on machine state during execution.

# 60s & 70s: Push Operations Value Mnemonic *δ α* Description

0x60 PUSH1 0 1 Place 1 byte item on stack.

***µ*****s**[0] ≡ *c*(***µ***pc +f1)

where *c*(*x*) ≡

*I***b**[*x*] if *x <* ll*I***b**ll

0 otherwise

|  |  |  |
| --- | --- | --- |
|  | | The bytes are read in line from the program code’s bytes array.  The function *c* ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian). |
| 0x61 | PUSH2 | 0 1 Place 2-byte item on stack. |

. . .

***µ*s**[0] ≡ ***c*** (***µ***pc + 1) *. . .* (***µ***pc + 2)

with ***c***(***x***) ≡ (*c*(***x***0)*, ..., c*(***x****x* −1)) with *c* as defined as above.

The bytes are right-aligned (takes the lowest significant place in big endian).

. .

0x7f PUSH32 0 1 Place 32-byte (full word) item on stack.

***µ*s**[0] ≡ ***c*** (***µ***pc + 1) *. . .* (***µ***pc + 32) where ***c*** is defined as above.

The bytes are right-aligned (takes the lowest significant place in big endian).

# 80s: Duplication Operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0x80 | DUP1 | 1 | 2 | Duplicate 1st stack item. |
| 0x81 | DUP2 | 2 | 3 | ***µ*s**[0] ≡ ***µ*s**[0]  Duplicate 2nd stack item. |
|  |  |  |  | ***µ*s**[0] ≡ ***µ*s**[1] |

. . . . .

0x8f DUP16 16 17 Duplicate 16th stack item.

***µ*s**[0] ≡ ***µ*s**[15]

# 90s: Exchange Operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0x90 | SWAP1 | 2 | 2 | Exchange 1st and 2nd stack items.  ***µ*s**[0] ≡ ***µ*s**[1]  ***µ*s**[1] ≡ ***µ*s**[0] |
| 0x91 | SWAP2 | 3 | 3 | Exchange 1st and 3rd stack items. |

. . .

***µ*s**[0] ≡ ***µ*s**[2]

***µ*s**[2] ≡ ***µ*s**[0]

. .

0x9f SWAP16 17 17 Exchange 1st and 17th stack items.

***µ*s**[0] ≡ ***µ*s**[16]

***µ*s**[16] ≡ ***µ*s**[0]

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 36

# a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate’s log series:

*A***l** ≡ *A***l** · (*I*a*,* **t***,* ***µ*m**[***µ*s**[0] *. . .* (***µ*s**[0] + ***µ*s**[1] − 1)]) and to update the memory consumption counter: ***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[1])

The entry’s topic series, **t**, differs accordingly:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Mnemonic** | *δ* | *α* | **Description** |
| 0xa0 | LOG0 | 2 | 0 | Append log record with no topics. |
| 0xa1 | LOG1 | 3 | 0 | **t** ≡ ()  Append log record with one topic. |
|  |  |  |  | **t** ≡ (***µ*s**[2]) |

. . . . .

0xa4 LOG4 6 0 Append log record with four topics.

**t** ≡ (***µ*s**[2]*,* ***µ*s**[3]*,* ***µ*s**[4]*,* ***µ*s**[5])

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 37

# f0s: System operations

**Value Mnemonic** *δ α* **Description**

0xf0 CREATE 3 1 Create a new account with associated code.

**i** ≡ ***µ*m**[***µ*s**[1] *. . .* (***µ*s**[1] + ***µ*s**[2] − 1)]

[*ζ*](#_bookmark43) ≡ ∅

(***σ****,* ***µ***g*, A, z,* **o**) ≡

[Λ(](#_bookmark44)***σ***∗*, A, I*a*, I*o*, L*(***µ***g)*, I*p*,* ***µ*s**[0]*,* **i***, I*e + 1*, ζ, I*w) if ***µ*s**[0] ***σ***[*I*a]b

 ***σ****,* ***µ***g*, A,* 0*,* () otherwise

∧ *I <* 1024e

***σ***∗ ≡ ***σ*** except ***σ***∗[*I*a]n = ***σ***[*I*a]n + 1

***µ*s**[0] ≡ *x*

where *x* = 0 if *z* = 0, i.e., the [contract creation process failed,](#_bookmark48) or *I*e = 1024

(the maximum call depth limit is reached) or ***µ*s**[0] *>* ***σ***[*I*a]b (balance of the caller is too low to fulfil the value transfer); and otherwise *x* = ADDR(*I*a*,* ***σ***[*I*a]n*, ζ,* **i**), the address of the newly created account [(86).](#_bookmark45)

***µ***i ≡ *M* (***µ***i*,* ***µ*s**[1]*,* ***µ*s**[2])

***µ*o** ≡ ()

Thus the operand order is: value, input offset, input size.

0xf1 CALL 7 1 Message-call into an account.

(***σ****, g, A, x,* **o**) ≡

[Θ(](#_bookmark47)***σ****, A , I*a*, I*o*, t, t, C*CALLGAS(***σ****,* ***µ****, A*)*,*

*I*p*,* ***µ*s**[2]*,* ***µ*s**[2]*,* **i***, I*e + 1*, I*w)

∗





**i** ≡ ***µ*m**[***µ*s**[3] *. . .* (***µ*****s**[3] + ***µ*s**[4] − 1)]

if ***µ*s**[2] ***σ***[*I*a]b ∧

*I*e *<* 1024

(***σ****, g, A,* 0*,* ()) otherwise

*n* ≡ min({***µ*s**[6]*,* ll**o**ll})

***µ*m**[***µ*s**[5] *. . .* (***µ*s**[5] + *n* − 1)] = **o**[0 *. . .* (*n* − 1)]

***µ*o** = **o**

***µ***g ≡ ***µ***g + *g*

***µ*s**[0] ≡ *x*

*A*∗ ≡ *A* except *A*∗**a** ≡ *A***a** ∪ {*t*}

*t* ≡ ***µ*** [1] mod 2160

**s**

***µ***i ≡ *M* (*M* (***µ***i*,* ***µ*s**[3]*,* ***µ*s**[4])*,* ***µ*s**[5]*,* ***µ*s**[6])

where *x* = 0 if the [code execution for this operation failed,](#_bookmark50) or if

***µ*s**[2] *>* ***σ***[*I*a]b (not enough funds) or *I*e = 1024 (call depth limit reached); *x* = 1 otherwise.

Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.

*C*CALL(***σ****,* ***µ****, A*) ≡ *C*GAfSCAP(***σ****,* ***µ****, A*) + *C*EXTRA(***σ****,* ***µ****, A*)

f≡

*C*

CALLGAS

(***σ****,* ***µ****, A*) ≡

*C*GASCAP(***σ****,* ***µ****, A*) + *G*callstipend if ***µ*s**[2] =

*C*GASCAP(***σ****,* ***µ****, A*) otherwise

0

*C*GASCAP

(***σ****,* ***µ****, A*) min{[*L*](#_bookmark141)(***µ***g − *C*EXTRA(***σ****,* ***µ****, A*))*,* ***µ*s**[0]} if ***µ***g ≥ *C*EXTRA(***σ****,* ***µ****, A*)

***µ*s**[0] otherwise

*C*EXTRA(***σ****,* ***µ***f*, A*) ≡ [*C*aaccess](#_bookmark140)(*t, A*) + *C*XFER(***µ***) + *C*NEW(***σ****,* ***µ***)

f≡

*C*

XFER

(***µ***) ≡

*G*callvalue if ***µ*s**[2] = 0

0 otherwise

*C*NEW

(***σ****,* ***µ***) *G*newaccount if DEAD(***σ****, t*) ∧ ***µ*s**[2] 0

0 otherwise

0xf2 CALLCODE 7 1 Message-call into this account with an alternative account’s code.

Exactly equivalent to CALL except:



(***σ****, g, A, x,* **o**) ≡

Θ(***σ****, A*∗*, I*a*, I*o*, I*a*, t, C*CALLGAS(***µ***)*,*

*I*p*,* ***µ*s**[2]*,* ***µ*s**[2]*,* **i***, I*e + 1*, I*w)



if ***µ*s**[2] ***σ***[*I*a]b ∧

*I*e *<* 1024

(***σ****, g, A,* 0*,* ()) otherwise

Note the change in the fourth parameter to the call [Θ](#_bookmark47) from the 2nd stack value ***µ*s**[1] (as in CALL) to the present address *I*a. This means that the recipient is in fact the same account as at present, simply that the code is overwritten.

0xf3 RETURN 2 0 Halt execution returning output data.

*H*RETURN(***µ***) ≡ ***µ*m**[***µ*s**[0] *. . .* (***µ*s**[0] + ***µ*s**[1] − 1)]

This has the effect of halting the execution at this point with output defined.

See section [9.](#_bookmark52)

***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[1])

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 38

0xf4 DELEGATECALL 6 1 Message-call into this account with an alternative account’s code, but

persisting the current values for *sender* and *value*.

Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is ***µ*s**[2]. As a result, ***µ*s**[3], ***µ*s**[4], ***µ*s**[5] and ***µ*s**[6] in the definition of CALL should respectively be replaced with ***µ*s**[2], ***µ*s**[3], ***µ*s**[4] and ***µ*s**[5]. Otherwise it is equivalent to CALL except:

 Θ(***σ****, A*∗*, I*s*, I*o*, I*a*, t, C*CALLGAS(***µ***)*,*

(***σ****, g, A, x,* **o**) ≡

*I ,* 0*, I ,* **i***, I*

+ 1*, I*

)

p

v

e

w

if *I*

*<* 1024

e

(***σ****, g, A,* 0*,* ()) otherwise



Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call [Θ.](#_bookmark47)

This means that the recipient is in fact the same account as at present, simply that the code is overwritten *and* the context is almost entirely identical.

|  |  |  |  |
| --- | --- | --- | --- |
| 0xf5 | CREATE2 | 4 | 1 Create a new account with associated code. Exactly equivalent to CREATE except:  The salt *ζ* ≡ ***µ*s**[3]. |
| 0xfa | STATICCALL | 6 | 1 Static message-call into an account. Exactly equivalent to CALL except:  The argument ***µ*s**[2] is replaced with 0.  The deeper argument ***µ*s**[3], ***µ*s**[4], ***µ*s**[5] and ***µ*s**[6] are respectively replaced with ***µ*s**[2], ***µ*s**[3], ***µ*s**[4] and ***µ*s**[5].  The last argument of [Θ](#_bookmark47) is ⊥. |
| 0xfd | REVERT | 2 | 0 Halt execution reverting state changes but returning data and remaining gas. The effect of this operation is described in [(143).](#_bookmark55)  For the gas calculation, we use the memory expansion function,  ***µ***i ≡ *M* (***µ***i*,* ***µ*s**[0]*,* ***µ*s**[1]) |
| 0xfe | INVALID | ∅ | ∅ Designated invalid instruction. |
| 0xff | SELFDESTRUCT | 1 | 0 Halt execution and register account for later deletion. |

*A***s** ≡ *A***s** ∪ {*I*a}

*A***a** ≡ *A***a** ∪ {*r*}



∅ if ***σ***[*r*] = ∅ ∧ ***σ***[*I*a]b = 0

***σ***[*r*] ≡

(***σ***[*r*]n*,* ***σ***[*r*]b + ***σ***[*I*a]b*,* ***σ***[*r*]**s***,* ***σ***[*r*]c) if *r*  = *I*a

(***σ***[*r*]n*,* 0*,* ***σ***[*r*]**s***,* ***σ***[*r*]c) otherwise



where *r* = ***µ*** [0] mod 2160

**s**

***σ***[*I*a]b = 0

*C*SELFDESTRUCT

(***σ****,* ***µ***) ≡ *G*

selfdestruct

f

+ 0 if *r* ∈ [*A***a**](#_bookmark39)

*G*coldaccountaccess otherwise

f

+ *G*newaccount if DEAD(***σ****, r*) ∧ ***σ***[*I*a]b = 0

0 otherwise

Appendix I. Genesis Block

The genesis block is 15 items, and is specified thus:

(331) 0256*,* KEC RLP () *,* 0160*, stateRoot,* 0*,* 0*,* 02048*,* 234*,* 0*,* 0*,* 3141592*, time,* 0*,* 0256*,* KEC (42) *,* ()*,* ()

Where 0256 refers to the parent hash, a 256-bit hash which is all zeroes; 0160 refers to the beneficiary address, a 160-bit hash which is all zeroes; 02048 refers to the log bloom, 2048-bit of all zeros; 234 refers to the difficulty; the transaction trie root, receipt trie root, gas used, block number and extradata are both 0, being equivalent to the empty byte array. The sequences of both ommers and transactions are empty and represented by (). KEC (42) refe rs to th e Keccak-256 hash of

a byte array of length one whose first and only byte is of value 42, used for the nonce. KEC RLP ()

hash of the ommer list in RLP, both empty lists.

value refers to the

The proof-of-concept series include a development premine, making the state root hash some value *stateRoot*. Also *time* will be set to the initial timestamp of the genesis block. The latest documentation should be consulted for those values.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 39

Appendix J. Ethash

* 1. **Definitions.** We employ the following definitions: Name Value Description

*J*wordbytes 4 Bytes in word.

*J*datasetinit 230 Bytes in dataset at genesis. *J*datasetgrowth 223 Dataset growth per epoch. *J*cacheinit 224 Bytes in cache at genesis. *J*cachegrowth 217 Cache growth per epoch.

|  |  |  |
| --- | --- | --- |
| *J*epoch | 30000 | Blocks per epoch. |
| *J*mixbytes | 128 | mix length in bytes. |
| *J*hashbytes | 64 | Hash length in bytes. |
| *J*parents | 256 | Number of parents of each dataset element. |
| *J*cacherounds | 3 | Number of rounds in cache production. |
| *J*accesses | 64 | Number of accesses in hashimoto loop. |

* 1. **Size of dataset and cache.** The size for Ethash’s cache **c** ∈ B and dataset **d** ∈ B depend on the epoch, which in turn depends on the block number.

(332) *E* (*H* ) = *H*i

epoch

epoch

i

*J*

The size of the dataset growth by *J*datasetgrowth bytes, and the size of the cache by *J*cachegrowth bytes, every epoch. In order to avoid regularity leading to cyclic behavior, the size must be a prime number. Therefore the size is reduced by a multiple of *J*mixbytes, for the dataset, and *J*hashbytes for the cache. Let *d*size = ll**d**ll be the size of the dataset. Which is calculated using

1. *d*size = *E*prime(*J*datasetinit + *J*datasetgrowth · *E*epoch − *J*mixbytes*, J*mixbytes)

The size of the cache, *c*size, is calculated using

1. *c*size = *E*prime(*J*cacheinit + *J*cachegrowth · *E*epoch − *J*hashbytes*, J*hashbytes)

f

1. *E*

prime

(*x, y*) = *x* if *x/y* ∈ N

*E*prime(*x* − 2 · *y, y*) otherwise

* 1. **Dataset generation.** In order to generate the dataset we need the cache **c**, which is an array of bytes. It depends on the cache size *c*size and the seed hash **s** ∈ B32.
     1. *Seed hash.* The seed hash is different for every epoch. For the first epoch it is the Keccak-256 hash of a series of 32 bytes of zeros. For every other epoch it is always the Keccak-256 hash of the previous seed hash:

1. **s** = *C*seedhash(*H*i)
2. *C*seedhash

i

(*H* ) = f**0**32 if *E*epoch(*H*i) = 0

With **0**32 being 32 bytes of zeros.

KEC(*C*seedhash(*H*i − *J*epoch)) otherwise

* + 1. *Cache.* The cache production process involves using the seed hash to first sequentially filling up *c*size bytes of memory, then performing *J*cacherounds passes of the RandMemoHash algorithm created by [Lerner [2014](#_bookmark85)]. The initial cache **c** , being an array of arrays of single bytes, will be constructed as follows.

We define the array **c***i*, consisting of 64 single bytes, as the *i*th element of the initial cache:

f

1. **c***i*

Therefore **c** can be defined as

= KEC512(**s**) if *i* = 0

KEC512(**c***i*−1) otherwise

1. **c** [*i*] = **c***i* ∀ *i < n*

1. *n* = *c*size

*J*hashbytes

The cache is calculated by performing *J*cacherounds rounds of the RandMemoHash algorithm to the initial cache **c**:

1. **c** = *E*cacherounds(**c** *, J*cacherounds)
2. *E*cacherounds(**x***, y*) =

*E*RMH(**x**) if *y* = 1

**x** if *y* = 0

*E*cacherounds(*E*RMH(**x**)*, y* − 1) otherwise

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 40

Where a single round modifies each subset of the cache as follows:

(343) *E*RMH(**x**) = *E*rmh(**x***,* 0)*, E*rmh(**x***,* 1)*, , E*rmh(**x***, n* − 1)

(344) *E*rmh(**x***, i*) = KEC512(**x**[(*i* − 1 + *n*) mod *n*] ⊕ **x**[**x**[*i*][0] mod *n*])

with **x** = **x** except **x**[*j*] = *E*rmh(**x***, j*) ∀ *j < i*

* + 1. *Full dataset calculation.* Essentially, we combine data from *J*parents pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset is then generated by a number of items, each *J*hashbytes bytes in size:

(345) **d**[*i*] = *E*datasetitem(**c***, i*) ∀ *i <*

*d*size

hashbytes

*J*

In order to calculate the single item we use an algorithm inspired by the FNV hash [(Glenn Fowler [1991])](#_bookmark97) in some cases as a non-associative substitute for XOR.

(346) *E* (**x***,* **y**) = (**x** · (0x01000193 ⊕ **y**)) mod 232

FNV

The single item of the dataset can now be calculated as:

1. *E*datasetitem(**c***, i*) = *E*parents(**c***, i,* −1*,* ∅)

f

1. *E*

parents

(**c***, i, p,* **m**) = *E*parents(**c***, i, p* + 1*, E*mix(**m***,* **c***, i, p* + 1)) if *p < J*parents − 2

*E*mix(**m***,* **c***, i, p* + 1) otherwise

fKEC512(**c**[*i* mod *c*size] ⊕ *i*) if *p* = 0

(349) *E*mix(**m***,* **c***, i, p*) =

**m***,* **c**[*E*

(*i* ⊕ *p,* **m**[*p* mod l*J*

J]) mod *c*

] otherwise

*E*

FNV

FNV

hashbytes

*/J*wordbytes

size

* 1. **Proof-of-work function.** Essentially, we maintain a “mix” *J*mixbytes bytes wide, and repeatedly sequentially fetch *J*mixbytes bytes from the full dataset and use the *E*FNV function to combine it with the mix. *J*mixbytes bytes of sequential access are used so that each round of the algorithm always fetches a full page from RAM, minimizing translation lookaside buffer misses which ASICs would theoretically be able to avoid.

If the output of this algorithm is below the desired target, then the nonce is valid. Note that the extra application of KEC at the end ensures that there exists an intermediate nonce which can be provided to prove that at least a small amount of work was done; this quick outer PoW verification can be used for anti-DDoS purposes. It also serves to provide statistical assurance that the result is an unbiased, 256 bit number.

The PoW-function returns an array with the compressed mix as its first item and the Keccak-256 hash of the concatenation of the compressed mix with the seed hash as the second item:

(350)

PoW(*H*n*, H*n*,* **d**) = {**m**c(KEC(RLP(*L*H(*H*n)))*, H*n*,* **d**)*,* KEC(**s**h(KEC(RLP(*L*H(*H*n)))*, H*n) + **m**c(KEC(RLP(*L*H(*H*n)))*, H*n*,* **d**))}

With *H*n being the hash of the header without the nonce. The compressed mix **m**c is obtained as follows:

*n*mix

1. **m**c(**h***,* **n***,* **d**) = *E*compress(*E*accesses(**d***,* **s**h(**h***,* **n**)*,* **s**h(**h***,* **n**)*,* −1)*,* −4)

*i*=0

The seed hash being:

1. **s**h(**h***,* **n**) = KEC512(**h** + *E*revert(**n**)) *E*revert(**n**) returns the reverted bytes sequence of the nonce **n**:
2. *E*revert(**n**)[*i*] = **n**[ll**n**ll − *i*]

We note that the “+”-operator between two byte sequences results in the concatenation of both sequences.

The dataset **d** is obtained as described in section [J.3.3.](#_bookmark158) The number of replicated sequences in the mix is:

1. *n*

mix

= *J*mixbytes

hashbytes

*J*

In order to add random dataset nodes to the mix, the *E*accesses function is used:

f

1. *E*

accesses

(**d***,* **m***,* **s***, i*) = *E*mixdataset(**d***,* **m***,* **s***, i*) if *i* = *J*accesses − 2

*E*accesses(*E*mixdataset(**d***,* **m***,* **s***, i*)*,* **s***, i* + 1) otherwise

1. *E*mixdataset(**d***,* **m***,* **s***, i*) = *E*FNV(**m***, E*newdata(**d***,* **m***,* **s***, i*)) *E*newdata returns an array with *n*mix elements:
2. *E* (**d***,* **m***,* **s***, i*)[*j*] = **d**[*E* (*i* ⊕ **s**[0]*,* **m**[*i* mod *J*mixbytes ]) mod *d*size*/J*hashbytes · *n*

newdata

FNV

*J*

wordbytes

*n*

mix

mix

+ *j*] ∀ *j < n*

mix

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER **BERLIN VERSION** 41

The mix is compressed as follows:

f

1. *E*

compress

(**m***, i*) = **m** if *i* ;;: ll**m**ll − 8

*E*compress(*E*FNV(*E*FNV(*E*FNV(**m**[*i* + 4]*,* **m**[*i* + 5])*,* **m**[*i* + 6])*,* **m**[*i* + 7])*, i* + 8) otherwise

Appendix K. Anomalies on the Main Network

K.1. **Deletion of an Account Despite Out-of-gas.** At block 2675119, in the transaction 0xcf416c536ec1a19ed1fb89e 4ec7ffb3cf73aa413b3aa9b77d60e4fd81a4296ba, an account at address 0x03 was called and an out-of-gas occurred during the call. Against the equation [(207),](#_bookmark120) this added 0x03 in the set of touched addresses, and this transaction turned ***σ***[0*x*03] into ∅.

Appendix L. List of mathematical symbols

Symbol Latex Command Description

\bigvee This is the least upper bound, supremum, or join of all elements operated on. Thus it is the greatest element of such elements [(Davey and Priestley [2002](#_bookmark88)]).