ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

DR. GAVIN WOOD CTO, ETHEREUM PROJECT GAVIN@ETHEREUM.ORG

ABSTRACT. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, not least Bitcoin. 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. Technological-rooted movements like Bitcoin 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, shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as namecoin adapted this original "currency application" of the technology into other applications allbeit rather simplistic ones.

Ethereum is a project that attempts to build the generalised technology; technology on which all transactionbased 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 unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. 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 incompatibility, incompetance, unwillingness, expense, uncertainty, inconvenience or corruption of existing legal systems. By specifying a state-change system through a rich and unambiguous 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 word. The incorruptibility of judgement, often difficult to find, comes naturally from a disinterested algorithmic interpreter. Transparency, through being about 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.

Overall, we wish to provide a system such that the user can be guaranteed that any other individuals, systems or organisations that they interact with, they can do so with absolute confidence in the possible outcomes and how they might come about.

2. The Blockchain Paradigm

Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some final state. It is this final state which we accept as the canonical "version" of the world of Ethereum. The state can include such information as account balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissable. 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. We define a valid state transitions as one which comes about through a transaction. Formally:

(1)
$$\mathbb{S}_{t+1} \equiv \Upsilon(\mathbb{S}_t, T)$$

where Υ is the Ethereum state transition function. In Ethereum, Υ , together with \mathbb{S} are considerably more powerful then any existing comparable system; Υ allows components to carry out arbitrary computation, while \mathbb{S} allows components to store arbitrary state between transactions.

Transactions are collated into blocks; we then chain blocks together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state (though do not store the final state itself—that would be far too big). They also punctuate the transaction series with incentives for nodes to *mine*. This incentivisation takes places as a statetransition 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 10.4.

Formally, we expand to:

(2)
$$\mathbb{S}_{t+1} \equiv \Pi(\mathbb{S}_t, B)$$

(3)
$$B \equiv (..., [T_0, T_1, ...])$$

(4)
$$\Pi(\mathbb{S}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\mathbb{S}, T_0), T_1)...)$$

Where Ω is the block-finalisation state transition function (a function that rewards a nominated party); *B* is this block, which includes a series of transactions amongst some other components; and Π is the block-level statetransition 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.

2.1. Value. In order to incentivise computation within the network, there needs to be an agreed method for transmitting value. To address this issue, Ethereum has an intrinsic currency, Ether, known also as ETH. 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 10^{18} Wei. There exist other subdenominations of Ether:

Multiplier	Name
10^{0}	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	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.

2.2. 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 block) 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.

This would mean that past a given point in time (block), multiple states of the system may coexist: some nodes believing 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 uncertainty 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 algorithm ?. This process is described in detail in section 9.

3. 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. 3.1. World State. The world state (*state*, see Appendix G), is a mapping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix C). Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree (trie, see Appendix D). The trie requires a simple database backend that maintains a mapping of bytearrays to bytearrays; 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.

The account state comprises the first two, and potentially the last two, of the following fields:

- **nonce:** A scalar value equal to the number of transactions sent from this address, or, in the case of contract accounts, the number of contractcreations made by this account.
- **balance:** A scalar value equal to the number of wei owned by this address.
- stateRoot: A 256-bit hash equal to the root node of a further trie structure that encodes the storage contents of the contract. Though a separate data structure, this trie is still stored using the same underlying state database. This trie takes the form as a simple mapping between domains of 256-bit values.
- **codeHash:** The hash of the EVM code of this contract—this is the code that gets executed should this address reveive a 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 corresponding hashes for later retrieval.

If the latter two fields are missing, the node represents a simple (non-contract) account: it maintains no state and executes no code on receipt of a transaction.

3.2. The Transaction. A transaction (formally, T) is a single cryptographically signed instruction sent by an actor external to Ethereum. An external actor can be a person (via a mobile device or desktop computer) or could be from a piece of automated software running on a server. There are two types of transactions: those which result in message calls and those which result in the creation of new contracts. Both types specify a number of common fields:

- **nonce:** A scalar value equal to the number of transactions sent by the sender; formally T_n .
- 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 contract's account; formally T_v .
- **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 topped-up later; formally T_q .
- to: The 160-bit address of the message call's recipient or the zero address for a contract-creation transaction; formally T_r .
- **v**, **r**, **s**: Three scalar values corresponding to the signature of the transaction and used to determine the sender of the transaction.
- Additionally, a contract-creation transaction contains:
 - **body:** An unlimited size byte array specifying the EVM-code for the contract, formally $T_{\rm b}$.
 - init: An unlimited size byte array specifying the EVM-code for the contract initialisation procedure, formally T_{i} .

Both **body** and **init** are EVM-code fragments; **body** is executed each time the contract account receives a message call (either through a transaction or due to the internal execution of code). **init** is a piece of code that is executed only once at contract creation and that gets discarded immediately thereafter.

Whereas a message call transaction contains:

data: An unlimited size byte array specifying the input data of the message call, formally T_{d} .

Appendix H specifies the function mapping transactions to the sender S, which happens through the ECDMA of the SECP-256k1 curve, using the hash of the transaction 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).

3.3. The Block. The block in Ethereum is the collection of relevant pieces of information (known as the block *header*), H, together with a set of transactions, \mathbf{T} , and a set of other block headers \mathbf{U} that are known to have a parent equal to the present block's parent's parent (such blocks are known as *uncles*). The block header contains several pieces of information:

- **parentHash:** The SHA3 256-bit hash of the parent block, in its entirety; formally H_p .
- **unclesHash:** The SHA3 256-bit hash of the uncles list portion of this block; formally H_u .
- **coinbase:** The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_b .
- stateRoot: The SHA3 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r .
- transactionsHash: The SHA3 256-bit hash of the transactions list portion of this block; formally H_t .
- **difficulty:** A scalar value corresonding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d .
- timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s .
- extraData: An arbitrary byte array containing data relavant to this block. Generally 256-bit or

less, but could be much larger in the genesis block. Not formally used.

nonce: A 256-bit hash which proves that a sufficient amount of computation has been carried out on this block; formally H_n .

The other two components in the block are simply a list of uncle block headers (of the same format as above) and a list of the transactions (as defined previously). Formally, we can refer to a block B:

$$(5) B \equiv (H, \mathbf{U}, \mathbf{T})$$

We can assert its validity if and only if it satisfies several conditions: It must be internally consistent with the uncle and transaction block hashes; the given transactions \mathbf{T} , when executed in order on the base state S result in a new state of the identity H_r :

(6) $SHA3(RLP(\mathbf{U})) \equiv H_u$

(7)
$$\wedge$$
 SHA3(RLP(**T**)) \equiv H_t

(8) $\wedge r(\Pi(\mathbb{S}, B)) \equiv H_r$

where r(S) is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of S with values encoded using RLP.

3.3.1. Block Header Validity. We define $P(B_H)$ to be the parent block of B, formally:

(9)
$$P(H) \equiv B' : SHA3(RLP(B')) = H_p$$

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

$$D(H) \equiv \begin{cases} 2^{22} & \text{if genesis block} \\ P(H)_{H_d} + \frac{P(H)_{H_d}}{1024} & \text{if } H_t < P(H)_{H_t} + 42 \\ P(H)_{H_d} - \frac{P(H)_{H_d}}{1024} & \text{otherwise} \end{cases}$$

 ${\cal H}_t$ is the timestamp of block ${\cal H},$ must fulfill the relation:

(10)

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

The nonce, H_n , must satisfy the relation:

(12)
$$\operatorname{PoW}(H, H_n) \leqslant \frac{2^{256}}{H_d}$$

Where PoW is the proof-of-work function (see section 10.4): this evaluates to an pseudo-random number cryptographically dependent on the parameters H and H_n . Given an approximately uniform distribution in the range $[0, 2^{256})$, 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 cannot propogate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because satisfaction of this requirement depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult, and, 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):

(13)
$$V(H) \equiv \operatorname{PoW}(H, H_n) \leq \frac{2^{256}}{H_1}$$

(14)
$$\wedge \quad H_d = D(H)$$

(15) $\wedge \quad H_t > P(H)_H,$

4. Gas and Payment

In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of gas (see Appendix B for the fees associated with various computation). Thus any given fragment of programmable computation (this includes creating contracts, making message calls, utilising and accessing contract storage and executing operations on the virtual machine) has a universally agreed cost in terms of gas.

Every transaction has a specific amount of gas associated 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 considered 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 trusted contracts, a relatively high gas limit may be set and left alone.

Any ether used to purchase gas that is not refunded is delivered to the *coinbase* 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 therefore 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 minimum 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.

5. TRANSACTION EXECUTION

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

- (1) The transaction signature is valid;
- (2) the transaction nonce is valid (equivalent to the sender account's current nonce);
- the gas limit is no smaller than the intrinsic gas, g₀, used by the transaction;
- (4) the sender account balance contains at least the $\cos t$, v_0 , required in up-front payment;

Formally, we consider the function Υ , with T being a transaction and S the state:

(16)
$$\mathbb{S}' = \Upsilon(\mathbb{S}, T)$$

Thus S' is the post-transactional state.

We define intrinsic gas g_0 , the amount of gas this transaction requires to be paid prior to execution, as follows:

(17)
$$g_0 \equiv \begin{cases} (|T_{\mathbf{i}}| + |T_{\mathbf{b}}|)G_{data} + G_{create} & \text{if } T_r = 0\\ |T_{\mathbf{d}}|G_{data} + G_{call} & \text{otherwise} \end{cases}$$

where $|T_{\mathbf{d}}|$, $|T_{\mathbf{i}}|$ and $|T_{\mathbf{b}}|$ are the sizes, in bytes, of the transaction's associated data, initialisation and body EVM-code, respectively and G is defined in Appendix B.

The up-front cost v_0 is calculated as:

(18)
$$v_0 \equiv T_g T_p + T_v$$

the validity is determined as:

(19)
$$S(T) \neq \varnothing$$

(20) $\land \mathbb{S}[S(T)] \neq \varnothing$
(21) $\land T_n = \mathbb{S}[S(T)]_{nonce}$

(22) $\wedge g_0 \leq T_g$

(23) $\wedge v_0 \leq \mathbb{S}[S(T)]_{balance}$

The transaction must not require more gas to be utilised in this block than the global block limit of gas, specified at 10^6 .

The execution of a valid transaction begins with an irrevocable change made to the state: the nonce of the account of the sender (S(T)) is incremented by one and the balance is reduced by the up-front cost, v_0 . The gas available for the proceeding computation, g, is defined as $T_g - g_0$. The computation, whether contract creation 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.

We define the rollback state \mathbb{S}_0 :

(24) $\mathbb{S}_0 \equiv \mathbb{S}$ except:

(25)
$$\mathbb{S}_0[S(T)]_{balance} \equiv \mathbb{S}[S(T)]_{balance} -$$

(26)
$$\mathbb{S}_0[S(T)]_{nonce} \equiv \mathbb{S}[S(T)]_{nonce} + 1$$

Evaluating \mathbb{S}_0 from \mathbb{S}_0 depends on the transaction type; either contract creation or message call; we define the pairing of post-execution provisional state (\mathbb{S}_P) and remaining gas (g'): (27)

c

$$(\mathbb{S}_P, g') \equiv \begin{cases} \Lambda(\mathbb{S}_0, S(T), T_o, g, T_p, T_v, T_{\mathbf{i}}, T_{\mathbf{b}}) & \text{if } T_r = 0\\ \Theta^*(\mathbb{S}_0, S(T), T_o, T_r, g, T_p, T_v, T_{\mathbf{d}}) & \text{otherwise} \end{cases}$$

where

$$(28) g \equiv T_g - g_0$$

Note we use Θ^* to denote the fact that only the first two components of the function's value are taken; the third represents the message-call's output value (a byte array) and is unused in the context of transaction evaluation.

After the message call or contract creation is processed, the state is finalised by refunding g', the remaining gas, to the sender at the original rate. The Ether for the gas that was actually used is given to the miner, whose address is specified as the coinbase of the present block B. So we define the final state \mathbb{S}' in terms of the provisional state \mathbb{S}_P :

(29) $\mathbb{S}' \equiv \mathbb{S}_P$ except

$$(30) \qquad \mathbb{S}[s]_{balance} \equiv \qquad \mathbb{S}_P[s]_{balance} + gT_p$$

(31)
$$\mathbb{S}'[m]_{balance} \equiv \mathbb{S}_P[m]_{balance} + (g - g')T_p$$

$$(32) mtextbf{m} \equiv B_{H_{U}}$$

In the case that s = m then we simply return the Ether back to the sender/miner, collapsing the exception into:

(33)
$$\mathbb{S}'[s]_{balance} \equiv \mathbb{S}_P[s]_{balance} + g$$

6. CONTRACT CREATION

There are number of intrinsic parameters used when creating a contract: sender (s), nonce (o), available gas (g), gas price (p), endowment (v) together with the two arbitrary length byte arrays, **i** for the initialisation EVM code and **b** for the EVM code of the body.

We define the creation function formally as the function Λ , which evaluates from these values, together with the state \mathbb{S}_0 to the tuple containing the new state, additional database entries and remaining gas (\mathbb{S}_P, g') , as in 5:

(34)
$$(\mathbb{S}_P, g') \equiv \Lambda(\mathbb{S}_0, s, o, g, p, v, \mathbf{i}, \mathbf{b})$$

The address of the new contract is defined as being the rightmost 160 bits of the SHA3 hash of RLP encoding of the structure [sender, nonce]. In the unlikely event that the address is already in use, it is treated as a bigendian integer and incremented by one until an unused address is arrived at. Thus we define the creation address function A:

(35)
$$A(s,n) \equiv a$$
 where:

(36)
$$a = \arg\min_{x} : x \ge a' \land \mathbb{S}_0[x] = \emptyset$$

$$(37) a' = \mathcal{B}_{96..255} (SHA3 (RLP(\langle s, n \rangle)))$$

where SHA3 is the SHA3 256-bit hash function, RLP is the RLP encoding function, $\mathcal{B}_{a..b}(X)$ evaluates to binary value containing the bits of indices in the range [a, b] of the binary data X and $\mathbb{S}_0[x]$ is the address state of x or \emptyset if none exists. Note we use one fewer than the sender's nonce value; we assert that we have incremented 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 account's nonce is initially defined as zero, the balance as the value passed, the storage as empty and the code hash as the SHA3 256-bit hash of the code, thus the mutated state becomes \mathbb{S}^* :

(38)
$$\mathbb{S}^* \equiv \mathbb{S}$$
 except:

(39)
$$\mathbb{S}^*[A(s,\mathbb{S}[s]_{nonce})] \equiv (0,v,\emptyset, SHA3(\mathbf{b}))$$

It is asserted that the state database will also change such that it defines the pair $(SHA3(\mathbf{b}), \mathbf{b})$.

Finally, the contract is initialised through the execution of the initialising EVM code **i** according to the execution model (see section 8). Code execution can effect several events that are not internal to the execution state: the contract's storage can be altered, further contracts can be created and further message calls can be made. As such, the code execution function Ξ evaluates to a tuple of the resultant state S' and available gas remaining g'. Code execution depletes gas; thus it may exit before the code has come to a natural halting state. In this exceptional case we say an Out-of-Gas exception has occured: The evaluated state is defined as being the empty set \emptyset and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation. The gas remaining should of course be zero. If the creation was conducted as the manifestation of a transaction, then this doesn't affect payment of the intrinsic cost: it is paid regardless.

If such an exception does not occur, then the remaining gas is refunded to the originator and the now-altered state is allowed to persevere. Thus formally, we may specify the new state and gas as \mathbb{S}' , g' where:

$$\begin{array}{rcl}
(40) & (\mathbb{S}^{**}, g') &\equiv \Xi(\mathbb{S}^{*}, g, I) \\
(41) & \mathbb{S}_{P} &\equiv \begin{cases} \mathbb{S}_{0} & \text{if } \mathbb{S}^{**} = \varnothing \\ \mathbb{S}^{**} & \text{otherwise} \end{cases} \\
(42) & I_{a} &\equiv a \\
(43) & I_{o} &\equiv s \\
(44) & I_{p} &\equiv p \\
(45) & I_{\mathbf{d}} &\equiv [] \\
(46) & I_{s} &\equiv s \\
(47) & I_{v} &\equiv v \\
(48) & I_{\mathbf{b}} &\equiv \mathbf{i} \end{cases} \\
\end{array}$$

 $I_{\mathbf{d}}$ evaluates to the empty array. I_H and I_n have no special treatment and are determined from the blockchain.

7. Message Call

In the case of executing a message call, several parameters are required: sender (s), transaction originator (o), recipient (r), available gas (g), value (v) and gas price (p) together with an arbitrary length byte array, **d**, the input data of the call. Aside from evaluating to a new state and additional database entries, message calls also have an extra component—the output data denoted by the byte array **o**. This is ignored when executing transactions, however message calls can be initiated due to VM-code execution and in this case this information is used.

(49)
$$(\mathbb{S}_P, g', \mathbf{o}) \equiv \Theta(\mathbb{S}, s, o, r, g, p, v, \mathbf{d})$$

We define S_1 , the rollback state as the original state but with the value transferred to the recipient:

(50)
$$\mathbb{S}_1 \equiv \mathbb{S}$$
 except:

(51)
$$\mathbb{S}_1[r]_{balance} \equiv \mathbb{S}_1[r]_{balance} + v$$

If the recipient account contains no code to be executed (i.e. it's not a contract), then we define the final state as being equivalent to the rollback state. In the case that the recipient contains code to be executed (i.e. that the account is a contract), then the contract's body code (identified as the fragment whose SHA3 hash is $S[r]_{code}$]) is executed according to the execution model (see section 8). Just as with contract creation, if the execution halts due to an exhausted gas supply, then no gas is refunded to the caller and the state is reverted to the point immediately prior to code execution.

			S*	* ot	herwise
(52)	\mathbb{S}'	\equiv	$\{ S_1$	if	$\mathbb{S}^{**} = \varnothing$
			S_1	if	$\mathbb{S}_1[r]_{code} = \emptyset$

(53)		$(\mathbb{S}^{**},g',\mathbf{o})$	≡	$\Xi(\mathbb{S}_1, g, I)$
(54)		I_a	≡	a
(55)		I_o	\equiv	0
(56)		I_p	≡	p
(57)		$I_{\mathbf{d}}$	\equiv	d
(58)		I_s	≡	s
(59)		I_v	\equiv	v
(60)	Let	$SHA3(I_{\mathbf{b}})$	=	$\mathbb{S}[r]_{code}$

We assert that the state database contains the entry

 $(SHA3(I_{\mathbf{b}}), I_{\mathbf{b}}).$

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

8.1. **Basics.** The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to facilitate the SHA3-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has an unlimited size. 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 in the state database. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in memory or storage, it is stored separately in a virtual ROM. It cannot be read directly; instead it exists only as a model for determining the next instruction to execute.

The machine can have exceptional execution for several reasons, including stack underflows, invalid instructions and divides by zero. These unambiguously and validly result in immediate halting of the machine with all state changes left intact. The one piece of exceptional execution that does not leave state changes intact is the out-of-gas (OOG) exception. Here, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) and which will deal with it separately.

8.2. 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. Most operations require a single gas fee to be paid for their execution; exceptions include SSTORE, SLOAD, CALL, CREATE, BALANCE and SHA3. Secondly, gas may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for CREATE and CALL. Finally, gas may be paid due to an increase in the usage of the memory.

Over a contract's execution, the total fee for memoryusage 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 previously indexed memory will certainly result is an addition memory usage fee. Due to this fee it's highly unlikely addresses will ever go above 32-bit bounds since at the present price of ether and default gas price, that would cost around US\$20M for the memory fee alone.

Storage fees have a slightly nuanced behaviour—to incentivise contracts to minimise storage use (which corresponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is waived; in fact, it is effectively paid up-front since the initial usage of a storage location costs twice as much as the normal usage.

More formally, given an instruction, it is possible to calculate the gas cost of executing it as follows:

- SHA3 costs G_{sha3} gas
- SLOAD costs G_{sload} gas
- BALANCE costs $G_{balance}$ gas
- SSTORE costs $d.G_{sstore}$ gas where:
 - -d=2 if the new value of the storage is nonzero and the old is zero;
 - d = 0 if the new value of the storage is zero and the old is non-zero;
 - -d = 1 otherwise.
- CALL costs G_{call} , though additional gas may be taken for the execution of contract body code.
- CREATE costs G_{create} , though additional gas may be taken for the execution of contract initialisation code.
- STOP costs G_{stop} gas
- All other operations cost G_{step} gas.

Additionally, when memory is accessed with MSTORE, MSTORE8, MLOAD, RETURN, SHA3, CREATE or CALL, the memory should be enlarged to the smallest multiple of words such that all addressed bytes now fit in it. See Appendix I for a rigourous definition of the EVM gas cost.

8.3. Execution Environment. In addition to the system state S, and the remaining gas for computation g, 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 originated this execution.
- *I_p*, the price of gas in the transaction that originated this execution.
- *I*_d, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.

(

- I_s , the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- *I_v*, the value 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 previous block.
- I_n , the block number of the current block.

The execution model defines the function Ξ , which can compute the resultant state \mathbb{S}' and the remaining gas g', given these definitions:

(61)
$$(\mathbb{S}', g') \equiv \Xi(\mathbb{S}, g, I)$$

8.4. 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, S and the machine state, M. Formally, we define it recursively with a function X. This uses an iterator function O (which defines the result of a single cycle of the state machine) together with functions Z which determines if the present state is an exceptional halting state of the machine and H, specifying the output data of the instruction if and only if the present state is a normal halting state of the machine. We consider the empty sequence (denoted [] to be non-equal to the empty set, denoted \emptyset).

(62)
$$\Xi(\mathbb{S}, g, I) \equiv X(\mathbb{S}, \mathbb{M}, I)$$

$$(63) \mathbb{M}_g \equiv g$$

$$(64) \qquad \qquad \mathbb{M}_{pc} \equiv 0$$

$$(65) \mathbb{M}_M \equiv [0,0,\ldots]$$

 $\begin{array}{ccc} (66) & \mathbb{M}_i &\equiv & 0 \\ (67) & \mathbb{M}_S &\equiv & \end{array}$

(68)
$$X(\mathbb{S}, \mathbb{M}, I) \equiv \begin{cases} (\mathbb{S}, \mathbb{M}, I, []) & \text{if } Z(\mathbb{S}, \mathbb{M}, I) \\ (O(\mathbb{S}, \mathbb{M}, I), \mathbf{o}) & \text{if } \mathbf{o} \neq \emptyset \\ X(O(\mathbb{S}, \mathbb{M}, I)) & \text{otherwise} \end{cases}$$

where

(

$$\mathbf{o} \equiv H(\mathbb{S}, \mathbb{M}, I)$$

The machine state \mathbb{M} is defined as the tuple (g, pc, M, i, T) which are the gas available, the program counter, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents \mathbb{M}_M are a series of zeroes of size 2^{256} .

For the ease of reading, the instruction mnemonics, written in smallcaps (e.g. ADD), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given in Appendix I.

For the purposes of defining Z, H and O, we define w as the current operation to be executed:

(70)
$$w \equiv \begin{cases} I_{\mathbf{b}}[\mathbb{M}_{pc}] & \text{if } \mathbb{M}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

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

8.4.1. Exceptional Halting. The exceptional halting function Z is defined as:

(71)
$$Z(\mathbb{S}, \mathbb{M}, I) \equiv \mathbb{M}_g < C(\mathbb{S}, \mathbb{M}, I) \lor \|\mathbb{M}_S\| < \delta_w$$

This states that the execution is in an exceptional halting state if there is insufficient gas or if there are insufficient stack items. The astute read will realise that this implies that no instruction can, through its execution, cause an exceptional halt.

8.4.2. Normal Halting. We define the normal halting function H: (72)

$$H(\mathbb{S}, \mathbb{M}, I) \equiv \begin{cases} H_{\text{RETURN}}(\mathbb{M}) & \text{if } w = \text{RETURN} \\ [] & \text{if } w \in \{\text{STOP}, \text{SUICIDE}\} \\ \varnothing & \text{otherwise} \end{cases}$$

The data-returning halt operation, RETURN, has a special function H_{RETURN} , defined in Appendix I.

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

$$O(\mathbb{S}, \mathbb{M}, I) \equiv (\mathbb{S}', \mathbb{M}', I)$$

(74)
$$\Delta \equiv \alpha_w - \delta_w$$

(75)
$$\|\mathbb{M}'_T\| \equiv \|\mathbb{M}_T\| + \Delta$$

(76)
$$\forall x \in [\alpha_w, \|\mathbb{M}'_T\|) : \mathbb{M}'_{T_x} \equiv \mathbb{M}_{T_{x+\Delta}}$$

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:

(77)
$$\mathbb{M}'_g \equiv \mathbb{M}_g - C(\mathbb{S}, \mathbb{M}, I)$$

(78)
$$\mathbb{M}'_{pc} \equiv \begin{cases} J_o(\mathbb{M}) & \text{if } w \in \{\text{JUMP}, \text{JUMPI}\} \\ \mathbb{M}_{pc} + p & \text{if } w \in [\text{PUSH1}, \text{PUSH32}] \\ \mathbb{M}_{pc} + 1 & \text{otherwise} \end{cases}$$

where p is the byte size of the push instruction, defined as:

(79)
$$p \equiv w - \text{PUSH1} + 2$$

In general, we assume the memory and system state don't change:

$$(80) \qquad \qquad \mathbb{M}'_M \equiv \mathbb{M}_M$$

(81) $\mathbb{M}'_i \equiv \mathbb{M}_i$

 $(82) S' \equiv S$

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix I, alongside values for α and δ and formal a description of the gas requirements.

9. 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 block, 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.

This scheme notably ignores so-called *stale* blocks: valid, mined blocks, which were propogated too late into the network and thus were beaten to network consensus by a sibling block (one with the same parent). Such blocks become more common as the network propogation time approaches the ideal inter-block time. However, by counting the computation work of stale block headers, we are able to do better: we can utilise this otherwise wasted computation and put it to use in helping to butress the more popular blockchain making it a stronger choice over less popular (though potentially longer) competitors.

This increases overall network security by making it much harder for an adversary to silently mine a canonical blockchain (which, it is assumed, would contain different transactions to the current consensus) and dump it on the network with the effect of reversing existing blocks and the transactions within.

In order to validate the extra computation, a given block B may include the block headers from any known uncle blocks (i.e. blocks whose parent is equivalent to the grandparent of B). Since a block header includes the nonce, a proof-of-work, then the header alone is enough to validate the computation done. Any such blocks contribute toward the total computation or *total difficulty* of a chain that includes them. To incentivise computation and inclusion, a reward is given both to the miner of the stale block and the miner of the block that references it.

Thus we define the total difficulty of block B recursively as:

$$(83) B_t \equiv B'_t + B_d + \sum_{U \in B_{\mathbf{U}}} U_d$$

$$(84) B' \equiv P(H)$$

$$(85) H \equiv B_H$$

As such given a block B, B_t is its total difficulty, B' is its parent block, B_d is its difficulty and B_U is its set of uncle blocks.

10. BLOCK FINALISATION

The process of finalising a block involves three stages:

- (1) Validate (or, if mining, determine) uncles.
- (2) Apply rewards.
- (3) Verify (or, if mining, compute a valid) state and nonce.

10.1. **Uncle Validation.** The validation of uncle headers means nothing more than verifying that each uncle header is both a valid header and satisfies the relation of uncle to

the present block. Formally:

(86)
$$\bigwedge_{U \in B_{\mathbf{U}}} V(U) \land P(U) = P(P(B_H)) \land P(B_H) \neq B$$

10.2. **Reward Application.** The application of rewards to a block involves raising the balance of the accounts of the coinbase address of the block and each uncle by a certain amount. We raise the block's coinbase account by R_b , the block reward, and the coinbase of each uncle by $\frac{7}{8}$ of that. Formally we define the function Ω :

(87)
$$\Omega(B,\mathbb{S}) \equiv \mathbb{S}':\mathbb{S}'=\mathbb{S}$$
 except:

(88)
$$\mathbb{S}'[B_{Hb}] = \mathbb{S}[B_{Hb}] + R_b$$

(89)
$$\forall U \in B_{\mathbf{U}}: \quad \mathbb{S}'[U_b] = \mathbb{S}[U_b] + \frac{7}{8}R_b$$

We define the block reward as 1500 Finney:

(90) Let
$$R_b = 1.5 * 10^{16}$$

10.3. State & Nonce Validation. We may now define the function, Γ , that maps a block *B* to its initiation state:

(91)
$$\Gamma(B) \equiv \begin{cases} \mathbb{S}_0 & \text{if } P(B_H) = \varnothing \\ \mathbb{S}_i : r(\mathbb{S}_i) = P(B_H)_{H_r} & \text{otherwise} \end{cases}$$

Here, $r(\mathbb{S}_i)$ means the hash of the root node of a trie of state \mathbb{S}_i ; this is stored in the state database since the trie is a mutable data structure.

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

(92)
$$\Phi(B) \equiv B': B' = B^*$$
 except:
(93) $B'_n = n: \operatorname{PoW}(B^*, n) < \frac{2^{256}}{H_d}$

(94)
$$B^* \equiv B$$
 except: $B'_r = r(\Pi(\Gamma(B), B))$

As specified at the beginning of the present work, Π is the state-transition function, which is defined in terms of Ω , the block finalisation function and Υ , the transactionevaluation function, both now well-defined:

(95)
$$\Pi(\mathbb{S}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\mathbb{S}, T_0), T_1)...)$$

(96)
$$B \equiv (H, [T_0, T_1, ...], \mathbf{U})$$

Thus the complete block-transition mechanism, less **PoW**, the proof-of-work function is defined.

10.4. Mining **Proof-of-Work.** The mining proof-ofwork (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination 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 difficulty). 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. In Bitcoin's case the task is the SHA256 hash function. While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work function 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 requires a lot of memory and 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.

More formally, the proof-of-work function takes the form of ${\tt PoW}:$

(97)
$$\operatorname{PoW}(H_{\mathfrak{R}}, n) < \frac{2^{256}}{H_d}$$

Where $H_{\mathbb{R}}$ is the new block's header H, but without the nonce component; H_d is the new block's difficulty value (i.e. the block difficulty from section 9).

As of the proof-of-concept (PoC) series of the Ethereum software, the proof-of-work function is simplistic and does not attempt to secure these goals. It will be described here for completeness.

10.4.1. *PoC Series.* For the PoC series, we use a simplified proof-of-work. This is not ASIC resistant and is meant merely as a placeholder. It utilises the bare SHA3 hash function to secure the block chain by requiring the SHA3 hash of the concatenation of the nonce and the header's SHA3 hash to be sufficiently low.

It is formally defined as PoW:

(98) $\operatorname{PoW}(H, n) \equiv \operatorname{BE}(\operatorname{SHA3}(\operatorname{SHA3}(\operatorname{RLP}(H_{\mathfrak{B}})) \circ n))$

where: $\operatorname{RLP}(H_{\mathbf{R}})$ is the RLP encoding of the block header H, not including the final nonce component; SHA3 is the SHA3 hash function accepting an arbitrary length series of bytes and evaluating to a series of 32 bytes (i.e. 256-bit); n is the nonce, a series of 32 bytes; \circ is the series concatenation operator; $\operatorname{BE}(X)$ evaluates to the value equal to X when interpreted as a big-endian-encoded integer.

10.4.2. *Release Series.* For the release series, we use a more complex proof-of-work. This has yet to be formally

defined, but involves two components; firstly that it concerns contract processing, and by extension the evaluation of programs on the EVM. Secondly that it concerns the utilisation of either the blockchain or the full state trie.

As an overview, the output of the function is based upon the system state, defined as the hash of the root node of the state trie. A set of transactions, pseudo-randomly determined from the nonce value and selected from the last N blocks is taken. N is large enough and the selection criteria are such that execution of the transactions requires some non-negligable amount of processing by the EVM. Whenever code is executed on the EVM, it is pseudorandomly (seeded again by the nonce) corrupted before alteration. Corruption could involve switching addresses with other transactions or rotating them through in the state trie (perhaps to the next address with the same order of magnitude of funds), rotating through instructions that have equivalent stack behaviour (e.g. swapping ADD for SUB or GT for EQ), or more destructive techniques such as randomly changing opcodes. This results in a problem that both require generalised computation hardware and is sequentially memory (and perhaps even disk) hard.

Any specialised hardware to perform this task could also be leveraged to speed up (and thus drive down costs) of general Ethereum transaction processing.

11. Implementing Contracts

11.1. **Data Feeds.** External server runs node. Creates & signs a transaction every minute containing new data. Sends to contract which knows to accept data only from this address. Allows polling of data from all other contracts (possibly for a microfee).

11.2. **Random Numbers.** Hash the block timestamp; for a series of pseudorandoms, take the previous, add some constant and hash the result.

For a more secure pseudo-random offering both parties agree on a number of random data feed contracts; these are concatenated along with the block timestamp and hashed to produce the first number in the series.

12. FUTURE DIRECTIONS

Rather than being hard-specified in the protocol, the block limit will become based upon some fixed ratio about a long-term moving average of actual gas usage in blocks, allowing it to slowly increase according to the network's ability to manage the transaction volume.

State database won't keep everything; maintains an age and eventually throws out nodes that are not recent enough or checkpoints.

Nodes in state trie that haven't sent/received a transaction in over X blocks could be thrown out, reducing Ether-leakage and reducing the growth of the state database.

13. Conclusion

This paper has introduced, discussed and formally defined the protocol of Ethereum.

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 are owned either by External Actors or intrinsically (as an identity) an Autonomous Object within Ethereum. If an Account identifies an Autonomous Object, then Ethereum will also maintain a Storage State particular to that Account. 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 virtual object existant only within the hypothetical state of Ethereum. Has an intrinsic address. Incorporated only as the state of the storage component of the VM.
- **Storage State:** The information particular to a given Autonomous Object that is maintained between the times that it runs.
- **Message:** Data (as a set of bytes) and Value (specified as Ether) that is passed between two Accounts in a perfectly trusted way, 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 an Autonomous Object, 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: Synonym for Autonomous Object used for non-technical audiences.

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 (ala 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 a contract's program code.

EVM Code: The bytecode that the EVM can natively execute.

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.

Appendix B. Fee Schedule

Name	Value	Description*
G_{step}	1	Default amount of gas to pay for execution cycle.
G_{stop}	0	Nothing paid for the STOP operation.
G_{sha3}	20	Paid for a SHA3 operation.
G_{sload}	20	Paid for a SLOAD operation.
G_{sstore}	100	Paid for a normal SSTORE operation (doubled of waived sometimes).
$G_{balance}$	20	Paid for a BALANCE operation.
G_{create}	100	Paid for a CREATE operation and for a contract creation transaction.
G_{call}	20	Paid for a CALL operation and for a message call transaction.
G_{memory}	1	Paid for every additional word when expanding memory.
G_{txdata}	5	Paid for every byte of data or code for a transaction.

APPENDIX C. RECURSIVE LENGTH PREFIX

This is a method for encoding arbitrarily structured binary data (byte arrays).

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.

Appendix E. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding a 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.

APPENDIX F. FORMAL SPECIFICATION OF STRUCTURES

Though RLP is data-agnostic, it does specify a canonical representation for integer quantities. It is big-endian with no leading zero-bytes. Thus for elements than feasibly can be stored as integers, it becomes important to specify whether they should adhere to this canonical representation or be left in some other (perhaps more 'native') format.

In the case of counts, balances, fees and amounts of wei, the canon-integer form must be used when storing in RLP. We call these INTs.

In the case of hashes (256-bit or 160-bit), user-visible strings and specialised byte-arrays (e.g. hex-prefix notation from the trie), they should be stored as unformatted byte-array data and not altered into some other form. We call these BINs.

When interpreting RLP data, clients are required to consider non-canonical INT fields in the same way as otherwise invalid RLP data and dismiss it completely.

```
Specifically:
  for the Block header:
Γ
  parentHash: BIN (256-bit),
  unclesHash: BIN (256-bit),
  coinbase: BIN (160-bit),
  stateRoot: BIN (256-bit)
  transactionsHash: BIN (256-bit),
  difficulty: INT,
  timestamp: INT,
  extraData: BIN (typically 256-bit),
  nonce: BIN (256-bit)
]
   (note: 'nonce', the last element, refers to a hash here and so is binary)
  for entries in the State trie for normal addresses:
Γ
  balance: INT,
  nonce: INT
٦
  and for contract addresses:
Ε
  balance: INT,
  nonce: INT,
  storageRoot: BIN (256-bit),
  codeHash: BIN (256-bit)
1
   (note: 'nonce', the second element, refers to a tx-count here and so is integer)
  for message call transactions:
Ε
  nonce: INT,
  value: INT,
  gasPrice: INT,
  gas: INT,
  recvAddr: BIN (160-bit),
  data: BIN (arbitrary length),
  v: INT,
  r: INT,
  s: INT
]
  or, for contract creation transactions,
Г
  nonce: INT,
  value: INT,
  gasPrice: INT,
  gas: INT,
  0: BIN (160-bit),
  code: BIN (arbitrary length),
  init: BIN (arbitrary length),
  v: INT,
  r: INT,
```

s: INT]

(note: 'nonce', the first element, refers to a tx-count here and so is integer)

The nonce in the transaction refers to the total amount of transactions sent from the address up until that moment in time.

for blocks, there are no immediate data field, but lists:

```
[
blockHeader: [...]
uncleList: [ [...], [...], ...]
txList: [ [...], [...], ...]
```

]

Uncle-blocks contain only the uncle's header.

Appendix G. The State DB

The state database is a mapping between SHA3-256 hashes (i.e. 256-bit binary fragments) and arbitrary byte data.

APPENDIX H. SIGNING TRANSACTIONS

The method of signing transactions is similar to the 'Electrum style signatures'; it utilises the SECP-256k1 curve.

APPENDIX I. VIRTUAL MACHINE SPECIFICATION

As previously specified in section 8, 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 S' M' such that:

$$(99) O(\mathbb{S}, \mathbb{M}, I) \equiv (\mathbb{S}', \mathbb{M}', I)$$

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

When a 32-byte machine datum is converted to and from as a 160-bit address or hash, the leftwards (high-order for BE) 20 bytes are used and the right most 12 are discarded or filled with zeroes.

The general gas cost function is defined as:

(100)
$$C(\mathbb{S}, \mathbb{M}) \equiv G_{memory}(\mathbb{M}'_{i} - \mathbb{M}_{i}) + \begin{cases} c_{\text{SSTORE}} & \text{if } o = \text{SSTORE} \\ G_{call} & \text{if } o = \text{CALL} \\ G_{create} & \text{if } o = \text{CREATE} \\ G_{sha3} & \text{if } o = \text{SHA3} \\ G_{sload} & \text{if } o = \text{SLOAD} \\ G_{balance} & \text{if } o = \text{SLOAD} \\ G_{stop} & \text{if } o = \text{STOP} \\ G_{step} & \text{otherwise} \end{cases}$$

where c_o is specified in the appropriate section below and C_M is the memory cost function, given as the product of G_{memory} and the maximum of 0 and the ceiling of the number of words in size that the memory must be over the current number of words, \mathbb{M}_i in order that all accesses reference valid memory whether for read or write; \mathbb{M}'_i is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

All arit	0s: Stop and Arithmetic Operations All arithmetic is modulo 2 ²⁵⁶ .					
Value	Mnemonic	δ	α	Description		
0x00	STOP	0	0	Halts execution. $\mathbb{M}'_R = []$		
0x01	ADD	2	1	Addition operation. $\mathbb{M}'_T[0] \equiv \mathbb{M}_S[0] + \mathbb{M}_S[1]$		
0x02	MUL	2	1	Multiplication operation. $\mathbb{M}'_T[0] \equiv \mathbb{M}_S[0] \times \mathbb{M}_S[1]$		
0x03	SUB	2	1	Subtraction operation. $\mathbb{M}'_T[0] \equiv \mathbb{M}_S[0] - \mathbb{M}_S[1]$		
0x04	DIV	2	1	Integer division operation. $\mathbb{M}'_T[0] \equiv \lfloor \mathbb{M}_S[0] \div \mathbb{M}_S[1] \rfloor$		
0x05	SDIV	2	1	Signed integer division operation. $\mathbb{M}'_T[0] \equiv \lfloor \mathbb{M}_S[0] \div \mathbb{M}_S[1] \rfloor$ Where all values are treated as signed 256-bit integers for the purposes of this operation.		
0x06	MOD	2	1	Modulo remainder operation. $\mathbb{M}'_T[0] \equiv \mathbb{M}_S[0] \mod \mathbb{M}_S[1]$		
0x07	SMOD	2	1	Modulo remainder operation. $\mathbb{M}'_T[0] \equiv \mathbb{M}_S[0] \mod \mathbb{M}_S[1]$ Where all values are treated as signed 256-bit integers for the purposes of this operation.		
0x08	EXP	2	1	Exponential operation. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{S}[0]^{\mathbb{M}_{S}[1]}$		
0x09	NEG	1	1	Negation operation. $\mathbb{M}'_T[0] \equiv -\mathbb{M}_S[0]$ Where all values are treated as signed 256-bit integers for the purposes of this operation.		
0x0a	LT	2	1	Less-than comparision. $\mathbb{M}'_{T}[0] \equiv \begin{cases} 1 & \text{if } \mathbb{M}_{S}[0] < \mathbb{M}_{S}[1] \\ 0 & \text{otherwise} \end{cases}$		
0x0b	GT	2	1	Greater-than comparision. $\mathbb{M}'_{T}[0] \equiv \begin{cases} 1 & \text{if } \mathbb{M}_{S}[0] > \mathbb{M}_{S}[1] \\ 0 & \text{otherwise} \end{cases}$		
0x0c	EQ	2	1	Equality comparision. $\mathbb{M}'_{T}[0] \equiv \begin{cases} 1 & \text{if } \mathbb{M}_{S}[0] = \mathbb{M}_{S}[1] \\ 0 & \text{otherwise} \end{cases}$		
0x0d	NOT	1	1	Simple not operator. $\mathbb{M}'_{T}[0] \equiv \begin{cases} 1 & \text{if } \mathbb{M}_{S}[0] = 0\\ 0 & \text{otherwise} \end{cases}$		
$\mathbb{M}_{S}[0]_{i}$	gives the i th bi	it (c	coun	10s: Bitwise Logic Operations ting from zero) of $\mathbb{M}_{S}[0]$		
Value	Mnemonic	δ	α	Description		
0x10	AND	2	1	Bitwise AND operation. $\forall i \in [0255] : \mathbb{M}'_T[0]_i \equiv \mathbb{M}_S[0]_i \land \mathbb{M}_S[1]_i$		
0x11	OR	2	1	Bitwise OR operation. $\forall i \in [0255] : \mathbb{M}'_T[0]_i \equiv \mathbb{M}_S[0]_i \lor \mathbb{M}_S[1]_i$		
0x12	XOR	2	1	Bitwise XOR operation. $\forall i \in [0255] : \mathbb{M}'_T[0]_i \equiv \mathbb{M}_S[0]_i \oplus \mathbb{M}_S[1]_i$		
0x13	BYTE	2	1	Retrieve single byte from word. $\forall i \in [07] : \mathbb{M}'_{T}[0]_{i} \equiv \begin{cases} \mathbb{M}_{S}[1]_{(i+8\mathbb{M}_{S}[0])} & \text{if } \mathbb{M}_{S}[0] < 32\\ 0 & \text{otherwise} \end{cases}$ For Nth byte, we count from the left (i.e. N=0 would be the most significant in big endiated of the second secon		

					20s: SHA3
Value	Mnemonic δ	5 0	α	De	escription
0x20	SHA3 2	2	1	\mathbb{M}'_{2} \mathbb{M}'_{i}	mpute SHA3-256 hash. $ \Gamma[0] \equiv SHA3(\mathbb{M}_M[\mathbb{M}_S[0] \dots (\mathbb{M}_S[0] + \mathbb{M}_S[1] - 1)]) $ $ \equiv \max(\mathbb{M}_i, \lceil (\mathbb{M}_S[0] + \mathbb{M}_S[1]) \div 32 \rceil) $
					30s: Environmental Information
Value	Mnemonic		δ	0	Description
0x30	ADDRESS		0	1	Get address of currently executing account. $\mathbb{M}'_T[0] \equiv I_a$
0x31	BALANCE		0	1	Get balance of currently executing account. $\mathbb{M}'_T[0] \equiv \mathbb{S}[I_a]_{balance}$
0x32	ORIGIN		0	1	Get execution origination address. $\mathbb{M}'_T[0] \equiv I_o$ This is the sender of original transaction; it's never a contract.
0x33	CALLER		0	1	Get caller address. $\mathbb{M}'_T[0] \equiv I_c$ 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. $\mathbb{M}'_T[0] \equiv I_v$
0x35	CALLDATALOA	4D	1	1	Get input data of current environment. $\mathbb{M}'_T[0] \equiv I_{\mathbf{d}}[\mathbb{M}_S[0] \dots (\mathbb{M}_S[0] + 32)]$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZI	E	0	1	Get size of input data in current environment. $\mathbb{M}'_T[0] \equiv I_{\mathbf{d}} $ This pertains to the input data passed with the message call instruction or transaction.
0x37	GASPRICE		0	1	Get price of gas in current environment. $\mathbb{M}'_{T}[0] \equiv I_{p}$ This is gas price specified by the originating transaction.
					40s: Block Information
Value	Mnemonic	δ	α	D	escription
0x40	PREVHASH	0	1	G M L	et hash of most recent complete block. $I'_T[0] \equiv I_{H_p}$ <i>i</i> is the previous block's hash.
0x41	COINBASE	0	1	G	et the block's coinbase address. $T'_{T}[0] \equiv I_{H_{b}}$
0x42	TIMESTAMP	0	1	G M	et the block's timestamp. $I'_{T}[0] \equiv I_{Ht}$
0x43	NUMBER	0	1	\mathbb{G} \mathbb{M} n	et the block's number. $I'_{T}[0] \equiv n(I_{H})$ (I_{H}) is the total number of ancestor blocks of the current block.
0x44	DIFFICULTY	0	1	G M	et the block's difficulty. $I'_{T}[0] \equiv I_{Hd}$
0x45	GASLIMIT	0	1	G M P	et the block's gas limit. $I'_{T}[0] \equiv 10^{6}$ resently, the gas limit is hard-coded in the protocol.

50s: Stack, Memory, Storage and Flow Operations					
Value	Mnemonic	δ	α	Description	
0x50	POP	1	0	Remove item from stack.	
0x51	DUP	1	2	Duplicate stack item. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{S}[0]$	
0x52	SWAP	2	2	Exchange stack items. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{S}[1]$ $\mathbb{M}'_{T}[1] \equiv \mathbb{M}_{S}[0]$	
0x53	MLOAD	1	1	Load word from memory. $ \mathbb{M}'_{T}[0] \equiv \mathbb{M}_{M}[\mathbb{M}_{S}[0] \dots (\mathbb{M}_{S}[0] + 32)] $ $ \mathbb{M}'_{i} \equiv \max(\mathbb{M}_{i}, \lceil (\mathbb{M}_{S}[0] + 32) \div 32 \rceil) $	
0x54	MSTORE	2	0	Save word to memory. $\mathbb{M}'_{M}[\mathbb{M}_{S}[0] \dots (\mathbb{M}_{S}[0] + 32)] \equiv \mathbb{M}_{S}[1]$ $\mathbb{M}'_{i} \equiv \max(\mathbb{M}_{i}, \lceil (\mathbb{M}_{S}[0] + 32) \div 32 \rceil)$	
0x55	MSTORE8	2	0	Save byte to memory. $\mathbb{M}'_{M}[\mathbb{M}_{S}[0]] \equiv (\mathbb{M}_{S}[1] \mod 256)$ $\mathbb{M}'_{i} \equiv \max(\mathbb{M}_{i}, \lceil (\mathbb{M}_{S}[0]+1) \div 32 \rceil)$	
0x56	SLOAD	1	1	Load word from storage. $\mathbb{M}'_{T}[0] \equiv \mathbb{S}'[I_{a}]_{storage}[\mathbb{M}_{S}[0]]$	
0x57	SSTORE	2	0	Save word to storage. $\mathbb{S}'[I_a]_{storage}[\mathbb{M}_S[0]] \equiv \mathbb{M}_S[1]$ $c_{\text{SSTORE}} \equiv \begin{cases} 2G_{sstore} & \text{if } \mathbb{M}_S[1] \neq 0 \land \mathbb{S}[I_a]_{storage}[\mathbb{M}_S[0]] = 0 \\ 0 & \text{if } \mathbb{M}_S[1] = 0 \land \mathbb{S}[I_a]_{storage}[\mathbb{M}_S[0]] \neq 0 \\ G_{sstore} & \text{otherwise} \end{cases}$	
0x58	JUMP	1	0	Alter the program counter. $J_{\text{JUMP}}(\mathbb{M}) \equiv \mathbb{M}_{S}[0]$ This has the effect of writing said value to \mathbb{M}_{pc} . See section 8.	
0x59	JUMPI	2	0	Conditionally alter the program counter. $J_{\rm JUMPI}(\mathbb{M}) \equiv \begin{cases} \mathbb{M}_S[0] & \text{if } \mathbb{M}_S[0] = 0 \\ \mathbb{M}_{pc} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to \mathbb{M}_{pc} . See section 8.	
0x5a	PC	0	1	Get the program counter. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{pc}$	
0x5b	MSIZE	0	1	Get the size of active memory. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{i}$	
0x5c	GAS	0	1	Get the amount of available gas. $\mathbb{M}'_{T}[0] \equiv \mathbb{M}_{g}$	
				60s & 70s: Push Operations	
Value	Mnemonic	δ	α	Description	
0x60	PUSH1	0	1	Place 1 byte item on stack. $\mathbb{M}'_{T}[0] \equiv I_{\mathbf{b}}[\mathbb{M}_{pc} + 1]$ The byte is right-aligned (takes the lowest significant place in big endian).	
0x61	PUSH2	0	1	Place 2-byte item on stack. $\mathbb{M}'_{T}[0] \equiv I_{\mathbf{b}}[(\mathbb{M}_{pc}+1)\dots(\mathbb{M}_{pc}+2)]$ 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. $\mathbb{M}'_{T}[0] \equiv I_{\mathbf{b}}[(\mathbb{M}_{pc}+1)\dots(\mathbb{M}_{pc}+32)]$	

16

	f0s: System operations						
Value	Mnemonic	δ	α	Description			
0xf0	CREATE	5	1	Create a new contract. $\mathbf{b} \equiv \mathbb{M}_{M}[\mathbb{M}_{S}[1] \dots (\mathbb{M}_{S}[1] + \mathbb{M}_{S}[2] - 1)]$ $\mathbf{i} \equiv \mathbb{M}_{M}[\mathbb{M}_{S}[3] \dots (\mathbb{M}_{S}[3] + \mathbb{M}_{S}[4] - 1)]$ $(\mathbb{S}', \mathbb{M}'_{g}) \equiv \Lambda(\mathbb{S}^{*}, I_{a}, I_{o}, \mathbb{M}_{g}, I_{p}, \mathbb{M}_{S}[0], \mathbf{i}, \mathbf{b})$ $\mathbb{S}^{*} \equiv \mathbb{S} \text{except} \mathbb{S}^{*}[I_{a}]_{nonce} = \mathbb{S}[I_{a}]_{nonce} + 1$ $\mathbb{M}'_{T}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to lack of gas; $x = A(I_{a}, \mathbb{S}[I_{a}]_{nonce})$, the address of the newly created contract, otherwise. $\mathbb{M}'_{i} \equiv \max(\mathbb{M}_{i}, [\max(\mathbb{M}_{S}[1] + \mathbb{M}_{S}[2], \mathbb{M}_{S}[3] + \mathbb{M}_{S}[4]) \div 32])$			
0xf1	CALL	7	1	$\begin{split} & \text{Message-call into contract.} \\ & \mathbf{i} \equiv \mathbb{M}_M[\mathbb{M}_S[3] \dots (\mathbb{M}_S[3] + \mathbb{M}_S[4] - 1)] \\ & \mathbf{o} \equiv \mathbb{M}_M[\mathbb{M}_S[5] \dots (\mathbb{M}_S[5] + \mathbb{M}_S[6] - 1)] \\ & \text{Let } g_k = \begin{cases} 0 & \text{if } \mathbb{M}_S[2] = 0 \\ \mathbb{M}_g - \min(\mathbb{M}_g, \mathbb{M}_S[2]) & \text{otherwise} \end{cases} \\ & (\mathbb{S}', g', \mathbf{o}) \equiv \Theta(\mathbb{S}, I_a, I_o, \mathbb{M}_S[0], g, I_p, \mathbb{M}_g - g_k, \mathbb{M}_S[1], \mathbf{i}) \\ & \mathbb{M}'_g \equiv g' + g_k \\ \mathbb{M}'_T[0] \equiv x \\ & \text{where } x = 0 \text{ if the code execution for this operation failed due to lack of gas;} \\ & x = 1 \text{ otherwise.} \\ & \mathbb{M}'_i \equiv \max(\mathbb{M}_i, [\max(\mathbb{M}_S[3] + \mathbb{M}_S[4], \mathbb{M}_S[5] + \mathbb{M}_S[6]) \div 32]) \end{split}$			
0xf2	RETURN	2	0	Halt execution returning output data. $H_{\text{RETURN}}(\mathbb{M}) \equiv \mathbb{M}_M[\mathbb{M}_S[0] \dots (\mathbb{M}_S[0] + \mathbb{M}_S[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 8. $\mathbb{M}'_i \equiv \max(\mathbb{M}_i, \lceil (\mathbb{M}_S[0] + \mathbb{M}_S[1]) \div 32 \rceil)$			
0xff	SUICIDE	1	0	Halt execution and obliterate account. $S'[\mathbb{M}_S[0]]_{balance} \equiv S[\mathbb{M}_S[0]]_{balance} + S[I_a]_{balance}$ $S'[I_a] \equiv \emptyset$			

Appendix J. Wire Protocol

The wire-protocol specifies a network-level protocol for how two peers can communicate. It includes handshake procedures and the means for transfering information such as blocks & transactions.

Appendix K. Genesis Block

The header of the genesis block is 9 items, and is specified thus:

 $(0_{256}, \text{SHA3}(\text{RLP}([])), 0_{160}, 0_{256}, \text{SHA3}(\text{RLP}([])), 2^{22}, 0, [], 42)$

Where:

 $\mathbf{0}_{256}$ refers to the parent hash, a 256-bit hash which is all zeroes.

 0_{160} refers to the coinbase address, a 160-bit hash which is all zeroes.

 2^{22} refers to the difficulty.

0 refers to the timestamp (the Unix epoch).

[] refers to the extradata, an empty byte array.

SHA3(RLP([])) values refer to the hashes of the transaction and uncle lists in RLP, both empty.