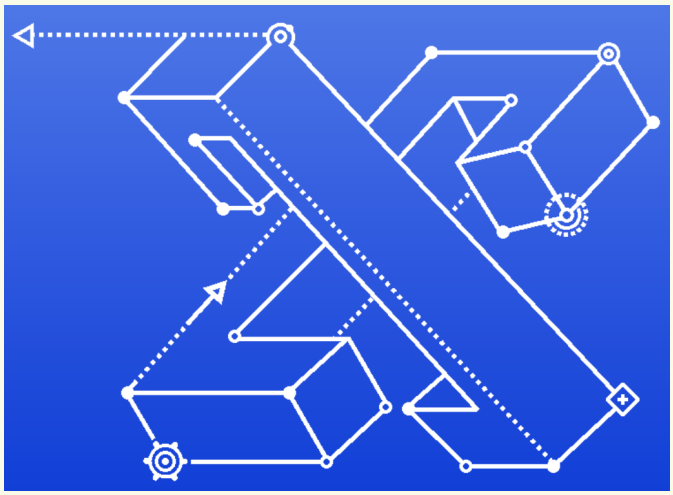
Conflux Protocol Specification

Chenxing Li[†], Guang Yang[†]

[†] *Conflux Dev Team*



Abstract

The success of Bitcoin and its follow-ups have demonstrated the value of decentralized consensus system among anonymous participants not trusting each other. On top of the consensus network there can be a public ledger or even a general state transition machine, such that all participants agree on the state of the ledger or the state machine. Conceptually the state machine can be Turing-complete and hence essentially a "world computer" shared by all participants, whose results cannot be tampered by any single person or entity. However, the processing power of the shared state machine is currently bottlenecked on the throughput of underlying consensus system.

Conflux implements a Turing-complete state machine on top of a high-throughput consensus network. To achieve a throughput of thousands of transactions per second, Conflux guarantees consensus on the total order of blocks organized in a Tree-Graph. In this way, all forked blocks contribute to the security and throughput of Conflux as well. In this work we discuss Conflux protocol design and implementation specifications.

Contents

1. Introduction

Since the born of Bitcoin, various blockchain projects have demonstrated extraordinary success with the power of consensus among permissionless and trustless parties. The most successful blockchain project after Bitcoin is widely considered to be Ethereum, which generalizes the blockchain paradigm from a specialized value-transfer system to a more generalized Turing-complete state machine that allows conceptually all kinds of computation. This generalized state machine, known as *Ethereum Virtual Machine* (EVM), makes the Ethereum network essentially a decentralized computing platform where the state advances on input of transactions. Sometimes Ethereum is referred to as the "world computer" that nobody can shut down, except that its processing power is rather poor and severely bounded by the throughput of underlying consensus.

The consensus throughput of Bitcoin is (in expectation) one block per 10 minutes, with block size 1MB (or 2MB with Segregated Witness (segwit)). Bitcoin is set to small block size and low generation rate mainly for security concerns. Intuitively, when there is no adversary, the natural probability of forks is proportional to the ratio of block broadcasting time to block (generation) time, since under the longest chain rule honest mining power may keep working on a fork during the propagation of a newly mined block. Ethereum applies a tailored version of GHOST rule [1] and smaller block size to achieve a much shorter block time, i.e. roughly $<$ 100KB per 15 seconds. Inclusive Block Chain Protocol [2] is a "block-DAG" proposal which defines a total order of blocks in a directed acyclic graph (DAG) rather than a chain, with the major advantage over GHOST that all forked blocks contribute to the consensus throughput as well. Another line of scaling techniques trades security and decentralization for scalability by using sharding, sidechains, or other second layer extensions. In extreme cases, centralized and somehow permissioned consensus systems are implemented in practice.

Conflux is a project which aims at building a high throughput first layer consensus system without any compromises in security and decentralization; a generalized computation platform that securely processes at least thousands of transactions per second which makes the throughput of consensus is no more a bottleneck. The positioning of Conflux is a strong backbone consensus network on which a numerous number of unprecedented applications and extensions can germinate and thrive. Technically, we follow a similar idea as [2] but organize blocks in a Tree-Graph, which enables a fast implementation of the Conflux protocol.

2. Conventions

Throughout this document, we use the following conventional notations:

- $\mathbb{B}$ denotes the set of bit values, i.e. $\mathbb{B} \equiv \{0,1\}$. $\mathbb{BY}$ denotes the set of byte values, i.e. $\mathbb{BY} \equiv \{0,\ldots,255\}$.
- $\mathbb{N}$ denotes the set of non-negative integers.
- For every $n \in \mathbb{N}$, we use $\mathbb{B}_n$ to denote a binary string of n bits, and $\mathbb{BY}_n$ for a string of n bytes. In particular, $\mathbb{BY} = \mathbb{B}_8$.
  - Furthermore, we denote by $\mathbb{B}^*$ and $\mathbb{BY}^*$ the set of binary or byte strings of arbitrary length, i.e. $\mathbb{B}^* \equiv \cup_{i\in\mathbb{N}}\mathbb{B}_i$ and $\mathbb{BY}^* \equiv \cup_{i\in\mathbb{N}}\mathbb{BY}_i$.
  - For convenience, we let $\mathbb{N}_n \equiv \{0,1,\ldots,2^n-1\}$ be the set of non-negative integers smaller than 2^n.
  - The empty string (or the empty series) is denoted by ε, which is distinguished from the empty set denoted by $\varnothing$.
- Numbers are in decimal base by default. Binary numbers are indicated with square brackets with subscripts, e.g. $[0100]_2$ is the 4-bit binary representation of 4. The subscript ch represents the character representation of bit string, e.g. $[\mathsf{ab}]_{\mathsf{ch}} = [6162]_{16} = [0110000101100010]_2$.
- When interpreting 256-bit binary values from $\mathbb{N}_{256}$ as integers, the representation is big-endian.
- When a 256-bit machine datum in $\mathbb{B}_{256}$ is converted to and from a 160-bit address or hash in $\mathbb{B}_{160}$, the rightwards (low-order for BE) 160 bits (20 bytes) are used and the leftmost 96 bits (12 bytes) are discarded or filled with zeros, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.
- Tuples are typically denoted with bold upper case letters such as $\mathbf{A}$. For frequently used tuples, we denote by T for a Conflux transaction, B for a Conflux block, and H for the header of a block.
  - Subscripts can be added to refer to an individual component in the tuple, e.g. T_n denotes the nonce of the transaction T. The type of referred components is the same as the type of subscript, e.g. B_H denotes the header of a block B, where the header itself is another tuple of elements. For succinctness we also write $\mathsf{H}(\mathsf{B}) \equiv \mathsf{B}_\mathsf{H}$ and sometimes simply H when there is no ambiguity. Also for succinctness, we sometimes use B and H interchangeably if there is no ambiguity, e.g. B_d stands for the target difficulty of block B, which is formally denoted as $\mathsf{H}(\mathsf{B})_d$ or $\mathsf{B}_{\mathsf{H}_d}$.
  - In case we are considering many transactions or blocks, we add superscripts to refer to a specific one of them, e.g. T^1 denotes the first transaction in a sequence.
- The Tree-Graph structure of blocks is typically denoted by $\mathbf{G}$, which is a graph consisting of blocks represented by vertices and parent/referee relations represented by two kinds of directed edges.

- Scalars and fixed-length sequences of elements (arrays, strings, and vectors) are denoted with normal lower case letters, e.g. n is used to denote a transaction nonce. Those with a special meaning may be denoted with Greek letters.
- Sequences of potentially arbitrary number of elements are typically denoted with bold lower case letters, e.g. $\mathbf{o}$ is used to denote the byte sequence generated as the output data of a message call.
- The highly structured state values are typically denoted with bold lower case Greek letters, e.g. $\boldsymbol{\sigma}$ (and its variants) is used to denote the world-state, and $\boldsymbol{\mu}$ for machine state.
- Square brackets are used to index subsequences of a sequence, with index starting from 0, e.g. $\boldsymbol{\mu}_\mathbf{s}[0]$ refers to the first item on the machine's stack and $\boldsymbol{\mu}_\mathbf{m}[0\ldots31]$ denotes the first 32 items of the machine's memory.
  - Some objects like the global state $\boldsymbol{\sigma}$ is interpreted as a set of key/value pairs. Thus the square bracket after $\boldsymbol{\sigma}$ refers to corresponding value of the given key (i.e., account address).
  - Square brackets use negative index to access an array from the end like Python, e.g. $\boldsymbol{\mu}_\mathbf{s}[-1]$ refers the last item on the machine's stack.
- Functions are typically denoted by upper case letters and subscripts are used for specialized variants, e.g. C is the general cost function and C_{CALL} is the cost function for the CALL operation. Specific functions operating on states are denoted by upper case calligraphic letters, e.g. $\mathscr{C}$ denotes the Conflux global state transition function.
  - For every function F defined on D, we let F^* denote the function on range D^* that element-wise applies F on its input items, i.e. $F^*\big((x_1,x_2,\ldots)\big) \equiv (F(x_1),F(x_2),\ldots)$.
- The superscript of a function with parentheses like $f^{(i)}$ represents call function f for i times recursively. Formally, $f^{(1)}(\cdot) \equiv f(\cdot)$ and $f^{(i)}(\cdot) \equiv f^{(i-1)}(f(\cdot))$ for $i \geq 2$.
- The indicator function $\mathbb{I}(\cdot)$ converts a boolean variable to integer. Formally, $\mathbb{I}(\mathsf{True}) = 1$ and $\mathbb{I}(\mathsf{False}) = 0$.

Frequently used functions:

- P: the *parent function* takes a block B as input and returns the parent block B′, i.e. B′ is referenced in B and designated as the parent of B. Formally, $\mathsf{P}(\mathsf{B}) \equiv \mathsf{B}' : \mathsf{KEC}\left(\mathsf{RLP}(\mathsf{B}')\right) = \mathsf{H}(\mathsf{B})_p$.[1]
- CHAIN: the *chain function* takes a block B as input and returns the chain from genesis block to block B following only parent edges. Formally, for genesis block G, $\mathsf{CHAIN}(\mathsf{G}) \equiv \mathsf{G}$; For other blocks B, $\mathsf{CHAIN}(\mathsf{B}) \equiv \mathsf{CHAIN}(\mathsf{P}(\mathsf{B})) \circ \mathsf{B}$.
- SIBLING: the *sibling function* takes a block B as input and returns the blocks which have the same parent as B. Formally, $\mathsf{SIBLING}(\mathsf{B}) \equiv \{\mathsf{B}' \mid \mathsf{P}(\mathsf{B}') = \mathsf{P}(\mathsf{B}) \wedge \mathsf{B}' \neq \mathsf{B}\}$.
- PAST: the *past function* takes a block B as input and outputs all blocks in the "past set of B", i.e. all blocks that are directly or indirectly referenced by B. The set $\mathsf{PAST}(\mathsf{B})$ does not include B, since B cannot reference itself.
- FUTURE: the *future function* takes a Tree-Graph $\mathbf{G}$ and a block $\mathsf{B} \in \mathbf{G}$ as inputs and outputs all blocks in the "future set of B". Formally, $\mathsf{FUTURE}(\mathsf{B};\mathbf{G}) \equiv \{\mathsf{B}' \in \mathbf{G} \mid \mathsf{B} \in \mathsf{PAST}(\mathsf{B}')\}$.
- EPOCH: the *epoch function* takes a block B as input and returns a sequence of all blocks in the epoch of B, sorted as in the Conflux total order defined in Section 4.2.4.
- BlockNo: the *block number function* takes a Tree-Graph $\mathbf{G}$ and a block B as inputs and returns the index of B in the total order of blocks specified by $\mathbf{G}$, where the index starts from 0. In particular, for the genesis block G and for every Tree-Graph $\mathbf{G}$ there must be $\mathsf{BlockNo}(\mathsf{G};\mathbf{G}) = 0$. Note that $\mathsf{BlockNo}(\mathsf{B};\mathbf{G})$ depends on $\mathbf{G}$ and it is different from $\mathsf{PAST}(\mathsf{B})$ which is fully determined by B. For every $\mathbf{G}$ and $\mathsf{B} \in \mathbf{G}$ there must be $\mathsf{BlockNo}(\mathsf{B};\mathbf{G}) \geq \mathsf{PAST}(\mathsf{B})$ since all blocks in $\mathsf{PAST}(\mathsf{B})$ precede B in the total order. Furthermore, for $\mathsf{B},\mathsf{B}' \in \mathbf{G}$ and $\mathsf{B} \neq \mathsf{B}'$, there must be $\mathsf{BlockNo}(\mathsf{B};\mathbf{G}) \neq \mathsf{BlockNo}(\mathsf{B}';\mathbf{G})$ because distinct blocks cannot have identical index in the total order. When the Tree-Graph $\mathbf{G}$ is clear from context, we may write $\mathsf{BlockNo}(\mathsf{B})$ for succinctness.
- PIVOT: the *pivot function* takes a block B as input and outputs the pivot block in the epoch of B.
- S: the *sender function* takes a transaction T as input and returns the sender of T, where in particular the sender is represented by its address.
- RLP: this is the serialization function that encodes an input of arbitrary length into a structured binary data, i.e. a byte array explicitly containing information about the length of the input. For more details see Appendix B in [3].
- ToList: this function takes a key/value set whose keys are integers or bit/byte sequences the values are integers. It outputs a sequence of key/value pairs for the entry with non-zero value in ascending order or lexicographical order of key.
- TRIE: the trie function maps an arbitrary-length binary byte array $\mathbf{s}$ into a 256-bit commitment that represents the database storing $\mathbf{s}$ in a modified Merkle Patricia tree (trie).
- KEC: the Keccak 256-bit cryptographic (collision-resistant) hash function that maps an arbitrary-length binary byte array to a random-looking binary string in $\mathbb{B}_{256}$. Furthermore, we assume KEC implements a *random oracle*, i.e. finding a random collision of KEC requires in expectation roughly 2^{128} attempts and a specific collision requires 2^{256}. Similarly, KEC512 refers to the 512-bit cryptographic hash function Keccak-512.

[1] One may argue whether P is well-defined since KEC has collisions. However, as long as the collision cannot be found in practical situations, the function P only need to look up such a B′ from existing blocks and returns $\perp$ if there is none.

- PoW: this is the proof-of-work function, which takes a block header as input and returns a scalar in $\mathbb{B}_{256}$.
- QUALITY: this is a measurement of the proof-of-work quality of a given block, i.e. how unlikely it is to find such a block. It takes a block or a block header as input and outputs a scalar in $\mathbb{N}_{256}$. Essentially, a block B with header $\mathsf{H} = \mathsf{H}(\mathsf{B})$ has quality $\mathsf{QUALITY}(\mathsf{B}) \equiv \mathsf{QUALITY}(\mathsf{H}) \equiv \left\lfloor 2^{256} / \left(\mathsf{PoW}(\mathsf{H}) - [\mathsf{H}_n[1\ldots127]]_2 \times 2^{128} + 1\right)\right\rfloor$ except for a few marginal cases. See Section 9.1 for more details.

2.1 Value

To incentivize the maintenance of the Conflux network and charge users for consumption of resources, Conflux has an intrinsic currency called Conflux Coin or simply Conflux, denoted by CFX for short. The smallest subdenomination is denoted by Drip, in which all values processed in Conflux are integers. One Conflux is defined as 10^{18} Drip. Frequently used subdenominations of Conflux are list as follows:

Multiplier (in Drip)	Name
10^0	Drip
10^9	GDrip
10^{12}	uCFX
10^{18}	Conflux (CFX)

3. Basic Components

In an overview, the Conflux world-state consists of a list of accounts and the associated account states, and the global state is updated via transactions. The Conflux blockchain stores all processed transactions in blocks, together with necessary information in block headers which enables a total ordering of all blocks. In this section we discuss the meaning of accounts, transactions and blocks in more details.

3.1 Accounts

The Conflux global state is described in an account model, with the basic storage component called an *account*. Every actor, which is either a person or an entity that is able to interact with the Conflux world, has its necessary information stored in an account α as a key/value pair $(\alpha_{addr}, \alpha_{state})$ of address and corresponding states. The account address α_{addr} is a 160-bit identifier. The account state $\alpha_{state} = (\alpha_{basic}, \alpha_{code}, \alpha_{storage}, \alpha_{deposit}, \alpha_{vote})$ consists of five components. The account basic state α_{state}, the code info α_{code}, the deposit list $\alpha_{deposit}$ and the vote list α_{vote} are four serialized sequences in an RLP structure (c.f. [3]). The account storage $\alpha_{storage}$ is a set of key-value pairs which map 256-bit address to a serialized storage information in an RLP structure. Furthermore we note that each account α is associated with a pair of public key and private key $\left(\alpha_{pubkey}, \alpha_{prikey}\right)$. The account address is the concatenation of 4-bit type indicator and a 156-bit digest of the associated public key:

$$\alpha_{addr} \equiv \mathsf{Type}_\alpha \circ \mathsf{KEC}\left(\alpha_{pubkey}\right)[100\ldots255] \tag{1}$$

where $\mathsf{Type}_\alpha \in \mathbb{B}_4$ is the type indicator, which is $\mathsf{Type}_\alpha = [0001]_2$ for *normal accounts* (a.k.a. *non-contract accounts*), $\mathsf{Type}_\alpha = [1000]_2$ for *(Solidity) contracts*, and $\mathsf{Type}_\alpha = [0000]_2$ for *built-in/reserved contracts* (a.k.a. *"precompiled contracts"*).

For succinctness and convenience, and as long as there is no ambiguity, we will write α without subscript for the state of an account and let $a \equiv \alpha_{addr}$ denote the corresponding address.

Basic state An account basic state $\alpha_{basic} \equiv (\alpha_n, \alpha_b, \alpha_c, \alpha_t, \alpha_o, \alpha_r, \alpha_a, \alpha_p)$ consists of the following fields.

- **nonce:** A scalar counter recording the number of previous activities initiated by this account. Formally denoted by $\alpha_n \in \mathbb{N}_{256}$. For example, the number of transactions sent from α_{addr}, or the number of contract-creations in the case this account is associated with codes.
- **balance:** A scalar value equal to the number of Drip owned by this account. Formally denoted by $\alpha_b \in \mathbb{N}_{256}$.
- **codeHash:** The hash of the EVM code that gets executed when α_{addr} receives a message call. Unlike other fields, it is immutable once established. All such code fragments are stored in a state database for later retrieval. This hash is formally denoted by $\alpha_c \in \mathbb{B}_{256}$, which satisfies $\alpha_c = \mathsf{KEC}(\mathbf{p})$ when the stored code is $\mathbf{p}$.
- **stakingBalance:** A scalar value equals to the number of staked Drip. Formally denoted by $\alpha_t \in \mathbb{N}_{256}$. (See section 8.3 for details)
- **storageCollateral:** A scalar value equals to the number of Drip used as collateral for storage, which will be returned to balance if the corresponding storage is released. Formally denoted by $\alpha_o \in \mathbb{N}_{256}$. (See section 7 for details)
- **accumulatedInterestReturn:** A scalar value equals to the number of Drip in accumulated interest return. Formally denoted by $\alpha_r \in \mathbb{N}_{256}$. (See section 8.3 for details)

- **admin:** The address of the administrator $\alpha_a \in \mathbb{B}_{160}$. (See section 8.2 for details)
- **sponsorInfo:** The sponsor information contains five components: **sponsor for gas**, **sponsor for collateral**, **sponsor gas limit**, **sponsor balance for gas** and **sponsor balance for collateral**. Formally denoted by $\alpha_p \in (\mathbb{B}_{160})^2 \times (\mathbb{N}_{256})^3$. (See section 8.1 for details). We use $\alpha_p[\mathsf{gas}]_a$, $\alpha_p[\mathsf{col}]_a$, $\alpha_p[\mathsf{limit}]_a$, $\alpha_p[\mathsf{gas}]_b$ and $\alpha_p[\mathsf{col}]_b$ to reference these five components.

Storage state An account's storage state $\alpha_{storage}$ (or $\alpha_\mathbf{s}$ for abbreviation) is a key/value database representing the account's storage state, if α is a contract account. Every storage entry is keyed by an arbitrary length bit-string $k \in \mathbb{B}^*$ and is represented as $(v,o) \in \mathbb{N}_{256} \times \mathbb{B}_{160}$, where v denotes the value stored in this entry and o denotes the owner who provides storage collateral for this entry. (See section 7 for details about storage collateral.)

We use $\alpha_\mathbf{s}[k]$ to represent storage entry (v,o) keyed by k and use $\alpha_\mathbf{s}[k]_v$ and $\alpha_\mathbf{s}[k]_o$ to denote corresponding fields. We denote by $\alpha_\mathbf{s}[k] = \varnothing$ for the case the storage state doesn't contain an entry with key k.

Code information For a contract account α, its code information $\alpha_{code} \equiv (\alpha_\mathbf{p}, \alpha_w)$ stores the account code $\alpha_\mathbf{p} \in \mathbb{BY}^*$ and the address $\alpha_w \in \mathbb{B}_{160}$ who paid the code storage collateral.

Staking vote info An account's staking vote list α_{vote} (or $\alpha_\mathbf{v}$ for abbreviation) is a key/value set representing the vote staking info unlock at given block number. Each entry is keyed by a block ordering index $i \in \mathbb{N}_{64}$ stores the staking vote amount $s \in \mathbb{N}_{256}$ in Drip. It means that the account promises to lock at least s Drip before block with order index i. See section 8.3 for more details.

Deposit list An account's deposit list $\alpha_{deposit} \in (\mathbb{N}_{256} \times \mathbb{N}_{64} \times \mathbb{N}_{256})^*$ is a series of deposit item. Each entry is a handful of deposited tokens in a ternary tuple of its amount (in Drip), its deposit time (measured in the total number of blocks) and its accumulated interest rate. We use $\alpha_{deposit}[\mathsf{amt}], \alpha_{deposit}[\mathsf{idx}], \alpha_{deposit}[\mathsf{accIR}]$ refers the three components. See section 8.3 for more details.

In initialization of each component, Conflux sets all the bit values, byte values and integers to zero and sets arbitrary length series to empty set, except the code hash which is set to the hash value of empty string. Each account state component is initialized as follows

$$\alpha^0_{basic} \equiv (0,0,\mathsf{KEC}(\varepsilon),0,0,0,0,(0,0,0,0,0)) \tag{2}$$
$$\alpha^0_{basic\_smp} \equiv (0,0,0,0,0) \tag{3}$$
$$\alpha^0_{code} \equiv (\varepsilon,0) \tag{4}$$
$$\alpha^0_{storage} \equiv \varepsilon \tag{5}$$
$$\alpha^0_{vote} \equiv \varepsilon \tag{6}$$
$$\alpha^0_{deposit} \equiv \varepsilon \tag{7}$$
$$\alpha^0 \equiv (\alpha^0_{basic}, \alpha^0_{code}, \alpha^0_{storage}, \alpha^0_{vote}, \alpha^0_{deposit}) \tag{8}$$

Given the Conflux world-state σ and an address $a \in \mathbb{B}_{160}$, we denote by $\sigma[a]$ for the state α_{state} of the account α with address $a = \alpha_{addr}$, i.e. $\sigma[a] \equiv \alpha_{state}$. We denote by $\sigma[a] \equiv \varnothing$ if the account with address a is never initialized. For succinctness, we denote by $\sigma[a]_\mathbf{s}$ for storage component $\alpha_{storage}$, $\sigma[a]_\mathbf{v}$ for staking vote component $\sigma[a]_{vote}$ and $\sigma[a]_\mathbf{d}$ for deposit component $\sigma[a]_{deposit}$.

The world-state σ never stores the account state α_{state} equals to initialization value α^0. So $\sigma[a] \equiv \varnothing$ is equivalent to $\sigma[a] \equiv \alpha^0$. For the key value structure components $\alpha_\mathbf{s}$ and $\alpha_\mathbf{v}$, we also use the notation $\varnothing$ and zero initialization value interchangeably.

3.2 Hash Digest of World-State

3.2.1 State entries

Conflux encodes each component of each account in format of $(k,v) \in \mathbb{B}^* \times \mathbb{B}^*$ respectively.

Basic entry. The basic entry stores basic components α_{basic} if $\alpha_{basic} \neq \alpha^0_{basic}$. Specially, for a non-contract address, Conflux doesn't stores the fields only related to contract, like code hash α_c, contract admin α_a and sponsor info α_p. Let $\alpha_{basic\_smp} \equiv (\alpha_n, \alpha_b, \alpha_t, \alpha_o, \alpha_r)$ denote the basic component with only the fields in a normal address. Formally

$$k \equiv \alpha_{addr} \tag{9}$$
$$v \equiv \begin{cases} \mathsf{RLP}(\alpha_{basic}) & \mathsf{Type}_\alpha = [1000]_2 \\ \mathsf{RLP}(\alpha_{basic\_smp}) & \text{otherwise} \end{cases} \tag{10}$$

Storage entry. The storage entry stores all the storages in $\alpha_{\mathbf{s}}$. Given account α with address α_{addr}, for each storage key $s \in \mathbb{B}_{256}$ with $\alpha_{\mathbf{s}}[s]_v \neq 0$, we have

$$k \equiv \alpha_{addr} \cdot [\mathsf{data}]_{\mathsf{ch}} \cdot s \tag{11}$$

$$v \equiv \begin{cases} \mathsf{RLP}(\alpha_{\mathbf{s}}[s]_v) & \alpha_{addr} = \alpha_{\mathbf{s}}[s]_o \\ \mathsf{RLP}(\alpha_{\mathbf{s}}[s]_v) & \alpha_{addr} = a_{\mathsf{stake}} \wedge k \in \{k_1, k_2, k_3, k_4, k_5\} \\ \mathsf{RLP}(\alpha_{\mathbf{s}}[s]) & \text{otherwise} \end{cases} \tag{12}$$

where: (13)

$$a_{\mathsf{stake}} \equiv \mathsf{0x0888000000000000000000000000000000000002} \tag{14}$$

$$k_1 \equiv [\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{15}$$

$$k_2 \equiv [\mathsf{interest\_rate}]_{\mathsf{ch}} \tag{16}$$

$$k_3 \equiv [\mathsf{total\_staking\_tokens}]_{\mathsf{ch}} \tag{17}$$

$$k_4 \equiv [\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{18}$$

$$k_5 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{19}$$

There are five special storage entries storing some statistic information about Conflux blockchain, which have no storage owner.

Storage root entry. The storage root entry stores the storage layout entry for account α if $\mathsf{Type}_{\alpha_{addr}} \in \{[0000]_2, [1000]_2\}$ and $\alpha_n \neq 0$.

$$k \equiv \alpha_{addr} \cdot [\mathsf{data}]_{\mathsf{ch}} \tag{20}$$

$$v \equiv [0000]_{16} \tag{21}$$

Code entry. The code entry stores the component code info α_{code} if $\alpha_{code} \neq \alpha^0_{code}$.

$$k \equiv \alpha_{addr} \cdot [\mathsf{code}]_{\mathsf{ch}} \cdot \alpha_c \tag{22}$$

$$v \equiv \mathsf{RLP}(\alpha_{code}) \tag{23}$$

Staking vote list entry. The staking vote list entry stores the component staking vote list $\alpha_{vote} \neq \alpha^0_{vote}$.

$$k \equiv \alpha_{addr} \cdot [\mathsf{vote}]_{\mathsf{ch}} \tag{24}$$

$$v \equiv \mathsf{RLP}(\mathsf{ToList}(\alpha'_{vote})) \tag{25}$$

where:

$$\alpha'_{vote} \equiv \alpha_{vote} \text{ removing all the key } x \text{ with } \alpha_{vote}[x] = \alpha_{vote}[x-1] \tag{26}$$

Deposit list entry. The deposit list entry stores the component deposit list $\alpha_{deposit} \neq \alpha^0_{deposit}$.

$$k \equiv \alpha_{addr} \cdot [\mathsf{deposit}]_{\mathsf{ch}} \tag{27}$$

$$v \equiv \mathsf{RLP}(\alpha_{deposit}) \tag{28}$$

3.2.2 Multi-version Merkle Patricia Trie

Ethereum collects all the key/value pairs into MPT (Merkle Patricia Trie) and updates it during execution of transactions. However, in a high throughput blockchain system, the MPT may be accessed in a high frequency and hence become a bottleneck of performance. Conflux maintains a **read-through write-back** cache and commits cached changes to MPT at the end of epoch execution.

Formally, at any time, Conflux maintains three key/value sets T_0, T_1, T_2, which are called **snapshot**, **intermediate set** and **delta set** respectively. The later one is the cache of the former one. T_2 is the only one updated during transaction execution. In order to balance the keys in MPT **delta set** and achieve a high performance, MPT T_2 has a different way in computing keys. Let

$$p \equiv \mathsf{KEC}(\mathsf{TRIE}(T_0), \mathsf{TRIE}(T_1)) \tag{29}$$

$$f_{addr}(\alpha_{addr}, p) \equiv \mathsf{KEC}(p[0..11] \cdot \alpha_{addr})[0..11] \cdot \alpha_{addr} \tag{30}$$

$$f_{store}(s, p) \equiv \mathsf{KEC}(p \cdot s)[4..31] \cdot s \tag{31}$$

(The index here are applied on byte-level.)

The key is defined as (the symbols are inherited from the previous section)

- **Basic entry.** $k \equiv f_{addr}(\alpha_{addr}, p)$
- **Storage entry.** $k \equiv f_{addr}(\alpha_{addr}, p) \cdot [\mathsf{data}]_{\mathsf{ch}} \cdot f_{store}(s, p)$
- **Storage root entry.** $k \equiv f_{addr}(\alpha_{addr}, p) \cdot [\mathsf{data}]_{\mathsf{ch}}$
- **Code entry.** $k \equiv f_{addr}(\alpha_{addr}, p) \cdot [\mathsf{code}]_{\mathsf{ch}} \cdot \alpha_c$
- **Staking vote list entry.** $k \equiv f_{addr}(\alpha_{addr}, p) \cdot [\mathsf{vote}]_{\mathsf{ch}}$
- **Deposit entry.** $k \equiv f_{addr}(\alpha_{addr}, p) \cdot [\mathsf{deposit}]_{\mathsf{ch}}$

Delta set. After an epoch execution, if a state entry is added, updated or removed, its newest content will be added to the delta set. Notice that the delta set only compares the state entry before and after the epoch execution. It is not aware of the intermediate steps during epoch execution. In case of deleting a state entry, the value of corresponding entry will be set to ε in the delta set.

The intermediate MPT will be merged to snapshot MPT every 100000 epochs. In particular, for some integer N, after execution of epoch $100000 \cdot N$, [2] Conflux will merge intermediate set T_1 to snapshot T_0 before computing the state root of pivot block on epoch $100000 \cdot N$. Each time the intermediate MPT is merged, Conflux sets current delta MPT as new intermediate set and resets delta set as emptyset. Formally,

$$T_0' \equiv \mathsf{MPTMerge}(T_0, T_1) \tag{32}$$
$$T_1' \equiv T_2 \tag{33}$$
$$T_2' \equiv \varnothing \tag{34}$$

where $\mathsf{MPTMerge}$ is the function updates the entries in T_0 which have a new version value in T_1. Specially, for the entries with initialization value, T_0 removes them.

3.2.3 State root

Let $\mathsf{TRIE}(T_0^{(i)}), \mathsf{TRIE}(T_1^{(i)}), \mathsf{TRIE}(T_2^{(i)})$ be the roots of MPT after execution of epoch i, then the state root of epoch i should be

$$StateRoot^{(i)} \equiv \mathsf{KEC}(\mathsf{RLP}(\mathsf{TRIE}(T_0^{(i)}), \mathsf{TRIE}(T_1^{(i)}), \mathsf{TRIE}(T_2^{(i)}))).$$

Notice that Conflux doesn't insert $StateRoot^{(i)}$ to the header of pivot block in epoch i, see the following for details.

3.3 Transactions

A Conflux transaction T is a single instruction composed by an external actor with a Conflux account α, and this instruction is cryptographically signed under the associated private key α_{prikey} of the sending account α. The authentication key, i.e. the sending account's associated public key α_{pubkey}, is also included in the transaction for verification.

There are two types of transactions depending on the "destinations":

1. to an account address: these are normal transactions that may transfer value and/or result in message calls, known as *action transactions*;
2. to "nowhere": these transactions are used to create new contracts, known as *contract creation transactions* or simply *creation transactions*.

Both types of transactions share the following common fields:

- **nonce:** A scalar value equal to the number of previously sent transactions. Formally denoted by $\mathsf{T}_n \in \mathbb{N}_{256}$.
- **gasPrice:** A scalar value indicating the number of Drip to be paid per unit of gas that is consumed as a result of the execution of T. Formally denoted by $\mathsf{T}_p \in \mathbb{N}_{256}$.
- **gasLimit:** A scalar value indicating the *total* amount of gas paid for the cost of the execution of T. This is paid up-front, before any actual computation is done, and may not be increased or refunded later. Formally denoted by $\mathsf{T}_g \in \mathbb{N}_{256}$. It is the transaction sender's responsibility to avoid any extravagance caused by an unnecessarily high **gasLimit**.
- **action:** A variable size field indicating the action of this transaction, which is either a **call** to an address or a **creation**, formally denoted by $\mathsf{T}_a \in \mathbb{B}_{160} \cup \mathbb{B}_0$. For a **call** transaction, action T_a indicates a 160-bit address of the recipient, which refers to either a normal account or a contract account; otherwise in case of a **creation** transaction, the recipient is indeed the newly created contract and we interpret T_a as the only element in $\mathbb{B}_0$ and write $\mathsf{T}_a = \varepsilon$.
- **value:** A scalar value equal to the amount of Drip that the transactions sender wants to transfer to the recipient, i.e. the account specified in T_a or the newly created contract. Formally denoted by $\mathsf{T}_v \in \mathbb{N}_{256}$.
- **storageLimit:** A scalar value indicating the maximum increment of storage used in the execution of T, measured in bytes. Formally denoted by $\mathsf{T}_\ell \in \mathbb{N}_{64}$.

[2] Since Conflux defers the execution of transactions for 5 epochs, the *execution of epoch i* means the execution for transactions in epoch $i-5$. See block component **deferredStateRoot** for details.

- **epochHeight:** A scalar value specifying the range of epochs where T can be executed. Formally denoted by $\mathsf{T}_e \in \mathbb{N}_{64}$ such that T can only be executed between the epochs of $[\mathsf{T}_e - 100000, \mathsf{T}_e + 100000]$.
- **chainID:** A binary chain id indicating where T is intended to be executed. Formally denoted by $\mathsf{T}_c \in \mathbb{N}_{32}$ and the chain id of Conflux is a constant $\mathsf{T}_c = 2$.
- **v, r, s:** Corresponding fields of the recoverable ECDSA signature of T, formally denoted by T_w, T_r and T_s.

In additional to the shared fields, a transaction may contain either of the following fields of unlimited length byte arrays for contract creation and invocation:

- **init:** A byte array specifying the EVM code for the initialization procedure, formally denoted by $\mathsf{T}_{\mathbf{i}} \in \mathbb{BY}^*$. Note that **init** is executed once and discarded thereafter, and it returns another code fragment **body** as the actual contract code that will be executed each time the contract account receives a message call (either through a transaction or due to the internal execution of code).
- **data:** A byte array specifying the input data of the message call to an existing contract, formally denoted by $\mathsf{T}_{\mathbf{d}} \in \mathbb{BY}^*$.

There is a function $\mathsf{S}(\cdot)$ that maps a transaction to its sender using the recoverable ECDSA signature, i.e. $\mathsf{S}(\mathsf{T})$ henceforth represents the sender of the transaction T. For convenience, we further introduce the function L_{T} that parses a transaction T as follows:

$$L_{\mathsf{T}}(\mathsf{T}) \equiv \begin{cases} (\mathsf{T}_n, \mathsf{T}_p, \mathsf{T}_g, \mathsf{T}_a, \mathsf{T}_v, \mathsf{T}_\ell, \mathsf{T}_e, \mathsf{T}_c, \mathsf{T}_{\mathbf{i}}, \mathsf{T}_w, \mathsf{T}_r, \mathsf{T}_s) & \text{if } \mathsf{T}_a = \varepsilon \\ (\mathsf{T}_n, \mathsf{T}_p, \mathsf{T}_g, \mathsf{T}_a, \mathsf{T}_v, \mathsf{T}_\ell, \mathsf{T}_e, \mathsf{T}_c, \mathsf{T}_{\mathbf{d}}, \mathsf{T}_w, \mathsf{T}_r, \mathsf{T}_s) & \text{otherwise} \end{cases} \tag{35}$$

3.4 Blocks

The Conflux blockchain organizes all on-chain information in blocks. Every Conflux block B consists of two parts: a block header H and a list of transactions Ts. The header H contains a list of other unreferenced block headers (a.k.a. *referee* blocks or simply *referees*).

The block header H is a collection of relevant pieces of information:

- **parentHash:** The Keccak 256-bit hash of the parent block's header, formally denoted by $\mathsf{H}_p \in \mathbb{B}_{256}$.
- **height:** A scalar value equal to the height of the block, which is also the number of parent references to reach the genesis block. This is formally denoted by $\mathsf{H}_h \in \mathbb{N}_{64}$. The genesis block has a height of zero.
- **timestamp:** A scalar value equal to the reasonable output of Unix's time() at this block's inception. Formally denoted by $\mathsf{H}_s \in \mathbb{N}_{64}$.
- **author:** The 160-bit address of the author of this block, formally denoted by $\mathsf{H}_a \in \mathbb{B}_{160}$. This is indeed the beneficiary's address to receive all rewards caused by successfully mining this block.
- **transactionRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transaction list portion of the block, formally $\mathsf{H}_t \in \mathbb{B}_{256}$.
- **deferredStateRoot:** The Keccak 256-bit hash commitment of the state after all "stable transactions" are executed and finalized, formally $\mathsf{H}_r \in \mathbb{B}_{256}$. Note that due to *deferred execution* in Conflux, "stable transactions" are those included in the past blocks of the pivot block of 5 epochs ago, i.e. 5 steps along the parent references.
  - If the **blame** field is zero, i.e. $\mathsf{H}_m = 0$, then H_r is the root node of the state trie after all "stable transactions" are executed and finalized.
  - Otherwise if $\mathsf{H}_m > 0$, then H_r will be the Keccak 256-bit hash of the vector consisting of the state root in the previous case and the corrected state roots of B's ancestors until (not including) the first ancestor that is not blamed.
- **deferredReceiptsRoot:** The Keccak 256-bit hash commitment of the receipts of each transaction executed when updating the **deferredStateRoot** field of the block, formally $\mathsf{H}_e \in \mathbb{B}_{256}$.
  - If the **blame** field is nonzero, i.e. $\mathsf{H}_m = 0$, then H_e is the Keccak 256-bit hash of the root node of the trie structure populated with the receipts of transactions in the epoch that is just executed.
  - Otherwise if $\mathsf{H}_m > 0$, then H_e will be the Keccak 256-bit hash of the vector consisting of the receipt root in the previous case and the corrected receipt roots of B's ancestors until (not including) the first ancestor that is not blamed.
- **deferredLogsBloomHash:** The Keccak 256-bit hash commitment of the Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipts of all transactions executed when updating the **deferredStateRoot** field, formally $\mathsf{H}_b \in \mathbb{B}_{256}$.
  - If the **blame** field is zero, i.e. $\mathsf{H}_m = 0$, then H_b is the Keccak 256-bit hash commitment of the Bloom filter composed from receipts of transactions in the epoch that is just executed.

  - Otherwise if $\mathsf{H}_m > 0$, then H_b will be the Keccak 256-bit hash of the vector consisting of the Bloom filter commitment in the previous case and the corrected Bloom filter commitments of B's ancestors until (not including) the first ancestor that is not blamed.

- **blame:** A scalar value $\mathsf{H}_m \in \mathbb{N}_{32}$ indicating the number of immediate ancestors whose header is incorrect in the execution state fields: **deferredStateRoot**, **deferredReceiptsRoot**, **deferredLogsBloomHash**, or **blame**. This **blame** field is relevant in reward distribution.
  For example, if $\mathsf{P}(\mathsf{B})$ is correct on all the four fields then $\mathsf{H}_m = 0$; if any of these fields is wrong in $\mathsf{P}(\mathsf{B})$, then $\mathsf{H}_m \geq 1$.
- **difficulty:** A scalar value $\mathsf{H}_d \in \mathbb{N}_{256}$ specifying the target difficulty level of this block. This is calculated from the previous block's difficulty level and the timestamp.
- **adaptiveWeight:** The Boolean value $\mathsf{H}_w \in \mathbb{B}$ indicating whether adaptive weight is triggered.
- **gasLimit:** A scalar value $\mathsf{H}_\ell \in \mathbb{N}_{256}$ equal to the current limit of gas expenditure per block. Starting from $\mathsf{H}_\ell(\mathsf{G}) \equiv 3 \times 10^7 = 30000000$.
- **refereeHash:** The serialized RLP sequence of the referee list consisting of Keccak 256-bit hashes of referee blocks, formally denoted by $\mathsf{H}_o \in \mathbb{B}\mathbb{Y}^*$. This list consists of up to 100 hash references of referee blocks. For convenience, we let $\mathsf{U}(\mathsf{B})$ denote these referee blocks of a block B.
- **customData:** A customized field with an arbitrary length list of byte string $\mathsf{H}_c \in (\mathbb{B}\mathbb{Y}^*)^*$.
- **nonce:** A 256-bit value which proves that a sufficient amount of computation has been carried out on this block, formally $\mathsf{H}_n \in \mathbb{B}_{256}$.

Formally, a block header is defined as

$$\mathsf{H} = (\mathsf{H}_p, \mathsf{H}_h, \mathsf{H}_s, \mathsf{H}_s, \mathsf{H}_a, \mathsf{H}_t, \mathsf{H}_r, \mathsf{H}_e, \mathsf{H}_b, \mathsf{H}_m, \mathsf{H}_d, \mathsf{H}_w, \mathsf{H}_\ell, \mathsf{H}_o, \mathsf{H}_c, \mathsf{H}_n) \tag{36}$$

The other part of B is simply a list of transactions. Therefore the block B can be represented as follows:

$$\mathsf{B} \equiv (\mathsf{B}_\mathsf{H}, \mathsf{B}_\mathsf{Ts}) \tag{37}$$

3.4.1 Transaction Receipt

For convenience and easy verification of the outcome of transaction execution, we introduce *transaction receipt* to record certain information of every executed transaction. When updating the **deferredStateRoot** H_r of a block, we encode a receipt $\mathsf{B}_\mathbf{R}[i]$ for the i-th executed transaction and store these receipts in an index-keyed trie. This root is recorded in the header as $\mathsf{H}_e \in \mathbb{B}_{256}$.

For every executed transaction T, the receipt $R \equiv \left(R_u, R_f, R_g, R_b, R_\mathbf{l}, R_z, R_\mathbf{o}, R_s, R_\mathbf{i}\right)$ is a tuple consisting of nine fields:

- $R_u \in \mathbb{N}_{256}$ is the cumulative gas used *in the epoch where* T *is executed* as of immediately after T has been processed;
- $R_f \in \mathbb{N}_{256}$ is the gas fee charged for the current transaction.
- $R_g \in \mathbb{B}$ denotes whether the gas fee is sponsored.
- $R_\mathbf{l}$ is the set of logs created in the execution of T;
- $R_b \in \mathbb{B}_{2048}$ is the Bloom filter composed from logs in $R_\mathbf{l}$;
- $R_z \in \mathbb{N}$ is the status code of the transaction T.
- $R_s \in \mathbb{B}$ denotes whether the storage collateral is sponsored.
- $R_\mathbf{o}$ is the set of incremental storage (in bytes) for addresses in the execution of T;
- $R_\mathbf{i}$ is the set of decremental storage (in bytes) for addresses in the execution of T;

The sequence $R_\mathbf{l} \equiv (O_0, O_1, \dots) \in \left(\mathbb{B}_{160} \times (\mathbb{B}_{256})^* \times \mathbb{B}_*\right)^*$ is a series of log entries, where each log entry O is a tuple of the logger's address $O_a \in \mathbb{B}_{160}$, a possibly empty series of 256-bit log topics $O_\mathbf{t} \equiv \left(O_{\mathbf{t}_0}, O_{\mathbf{t}_1}, \dots\right)$, such that $O_{\mathbf{t}_i} \in \mathbb{B}_{256}$ for every $i \in \mathbb{N}$, and a sequence of data $O_\mathbf{d} \in \mathbb{B}_*$:

$$O \equiv (O_a, O_\mathbf{t}, O_\mathbf{d}) \tag{38}$$

The Bloom filter function M reduces a log entry into a single 256-byte (2048-bit) hash as follows:

$$M(O) \equiv \bigvee_{x \in \{O_a\} \cup O_\mathbf{t}} (M_{3:2048}(x)) \tag{39}$$

where $M_{3:2048}$ is the specialized Bloom filter that sets three out of 2048 bits to 1 on input of an arbitrary byte sequence, as formally defined in [3].

The sequence $R_\mathbf{o} \equiv (P_0, P_1, \dots) \in (\mathbb{B}_{160} \times \mathbb{B}_{256})^*$ where each entry P is a tuple of an address and the incremental storage collateral of such address. $R_\mathbf{o}$ only collects the addresses with positive incremental storage collateral. It is sorted in ascending order of addresses and each address only appears once. $R_\mathbf{i}$ has the same settings as $R_\mathbf{o}$.

3.4.2 Serialization

The function L_B and L_H are the preparation functions for a block and block header respectively (similar as L_T defined in eq. (35)), where we recall that L^*_T and L^*_H refer to element-wise sequence transformations. We assert the types and order of the structure when the RLP transformation is required:

$$L_H(\mathsf{H}) \equiv (\mathsf{H}_p, \mathsf{H}_h, \mathsf{H}_s, \mathsf{H}_s, \mathsf{H}_a, \mathsf{H}_t, \mathsf{H}_r, \mathsf{H}_e, \mathsf{H}_b, \mathsf{H}_m, \mathsf{H}_d, \mathsf{H}_w, \mathsf{H}_\ell, \mathsf{H}_o, \mathsf{H}_c[0], \cdots, \mathsf{H}_c[-1], \mathsf{H}_n) \tag{40}$$

$$L_B(\mathsf{B}) \equiv (L_H(\mathsf{B}_\mathsf{H}), L^*_\mathsf{T}(\mathsf{B}_\mathsf{Ts})) \tag{41}$$

In addition, we let L_O be the preparation function for the referee blocks as follows:

$$L_O(\mathsf{H}) \equiv L^*_H(\mathsf{H}_o)$$

The component types are defined thus:

$$\begin{array}{lllllllllll} & \mathsf{H}_p \in \mathbb{B}_{256} & \wedge & \mathsf{H}_h \in \mathbb{N}_{64} & \wedge & \mathsf{H}_s \in \mathbb{N}_{64} & \wedge & \mathsf{H}_a \in \mathbb{B}_{160} & \wedge & \mathsf{H}_t \in \mathbb{B}_{256} \\ \wedge & \mathsf{H}_r \in \mathbb{B}_{256} & \wedge & \mathsf{H}_e \in \mathbb{B}_{256} & \wedge & \mathsf{H}_b \in \mathbb{B}_{256} & \wedge & \mathsf{H}_m \in \mathbb{N}_{32} & \wedge & \mathsf{H}_d \in \mathbb{N}_{256} \\ \wedge & \mathsf{H}_w \in \mathbb{B} & \wedge & \mathsf{H}_\ell \in \mathbb{N}_{256} & \wedge & \mathsf{H}_o \in (\mathbb{B}_{256})^{\leq 100} & \wedge & \mathsf{H}_n \in \mathbb{B}_{256} & & \end{array} \tag{42}$$

Now we have the specification for the construction of a formal block structure. With the RLP transformation we can further serialize this structure into a sequence of bytes ready for transmission and storage.

3.4.3 Well-formedness

Every Conflux block B (with header $\mathsf{H} = \mathsf{H}(\mathsf{B})$) is *well-formed* if and only if it is internally consistent and satisfies the following:

$$\mathsf{H}_t = \mathsf{TRIE}(\forall i < ||\mathsf{B}_\mathsf{Ts}||, i \in \mathbb{N} : (\mathsf{RLP}(i), \mathsf{RLP}(L_\mathsf{T}(\mathsf{B}_\mathsf{Ts}[i])))) \tag{43}$$

Intuitively, a block B is well-formed if its header H is consistent with the contents inside B. In other words, H effectively represents the whole block B.

3.4.4 Block Header Validity

Given a block B with header $\mathsf{H} = \mathsf{H}(\mathsf{B})$, we decide whether the header H is valid by checking the following fields of H and comparing to $\mathsf{H}\big(\mathsf{P}(\mathsf{B})\big)$ and $\mathsf{PAST}(\mathsf{B})$ when necessary:

- the height is increased by one;
- the timestamp (in Unix's time()) is increased;
- the canonical gas limit does not change too much (i.e. more than $1/1024$) and it remains above 10^7;
- the target difficulty is properly set according to Section 9.2;
- the proof-of-work quality exceeds the target difficulty;
- the parent is chosen properly from $\mathsf{PAST}(\mathsf{B})$ (the past view of B) following the GHAST rule;
- the adaptive weight flag **adaptiveWeight** must set properly according to the GHAST rule with respect to $\mathsf{PAST}(\mathsf{B})$;
- the referee list **refereeHash** properly decomposes to block headers.

Formally, the header H is valid if and only if:

$$\mathsf{H}_h = \mathsf{H}(\mathsf{P}(\mathsf{B}))_h + 1 \tag{44}$$

$$\wedge \quad \mathsf{H}_s > \mathsf{H}(\mathsf{P}(\mathsf{B}))_s \tag{45}$$

$$\wedge \quad |\mathsf{H}_l - \mathsf{H}(\mathsf{P}(\mathsf{B}))_l| < \left\lfloor \frac{\mathsf{H}(\mathsf{P}(\mathsf{B}))_l}{1024} \right\rfloor \quad \wedge \quad \mathsf{H}_l \geq 10^7 \tag{46}$$

$$\wedge \quad \mathsf{H}_d \text{ is legitimate according to the difficulty adjusting function} \tag{47}$$

$$\wedge \quad \mathsf{QUALITY}(\mathsf{H}) \geq \mathsf{H}_d \tag{48}$$

$$\wedge \quad \mathsf{P}(\mathsf{B}) \text{ specified by } \mathsf{H}_p \text{ is legitimate according to GHAST rule in } \mathsf{PAST}(\mathsf{B}) \tag{49}$$

$$\wedge \quad \mathsf{H}_w \text{ is legitimate according to the GHAST rule in } \mathsf{PAST}(\mathsf{B}) \tag{50}$$

$$\wedge \quad \mathsf{H}_o \text{ is well-formed and encodes referee block headers} \tag{51}$$

We remark that the validity of H only matters for consensus of total order of blocks, and it *does not* rely on the correctness of the execution related fields **deferredStateRoot**, **deferredReceiptsRoot**, **deferredLogsBloomHash** and **blame** of the header, i.e. $\mathsf{H}_r, \mathsf{H}_e, \mathsf{H}_b$ and H_m of H respectively. However, being incorrect in these fields may indicate that the author of the block fails to maintain the state and execute the transactions properly, in which case we still count the contribution of this block to consensus but the author gets no reward, as discussed in Section 3.4.6 and Section 10.

3.4.5 Partially (In)Valid Blocks

We call a block B *partially valid* if either $\mathsf{P}(\mathsf{B})$ is marked as partially valid or its header $\mathsf{H} = \mathsf{H}(\mathsf{B})$ passes all the assertions as in Section 3.4.4 except for the following:

- the parent reference $\mathsf{P}(\mathsf{B})$ specified by $\mathsf{H}(\mathsf{B})_p$ is not chosen properly according to the GHAST rule in $\mathsf{PAST}(\mathsf{B})$;
- the adaptive weight flag $\mathsf{H}(\mathsf{B})_w$ is not set properly according to the GHAST rule in $\mathsf{PAST}(\mathsf{B})$;
- the target difficulty $\mathsf{H}(\mathsf{B})_d$ satisfies the threshold condition of difficulty adjustment but it is not calculated properly according to the GHAST difficulty adjusting function.

A partially valid block may not be referenced by honest blocks in several hours after it is released. When a block becomes old and loses the ability to influence the pivot chain, we do not care about whether a block is partially valid or not. See Section 4.1 for details. Once a partially valid block is accepted, then it is treated as a fully valid block except for the decision of timer chain (Section 4.2.1).

We note that as long as the target difficulty is legitimate and the proof of work is valid, the partially valid block can still contribute to the throughput. This is because we allow referencing partially valid blocks and the transactions inside will be processed as in any fully valid block. We further remark that since the producer of a partially valid block is entitled to no reward, transaction fees may be burnt in case these transactions are only collected in partially valid blocks.

3.4.6 Blaming Mechanism

The **blame** field is introduced for easy verification of states by light nodes. Intuitively, this field represents the vote to the latest ancestor block which commits to a correct state, i.e. the author of the current block agrees with the committed state (represented in **deferredStateRoot**, **deferredReceiptsRoot**, **deferredLogsBloomHash**, and **blame**) of that block. We emphasize that these fields are *not* checked for validity of a block, however, committing to an incorrect state may lead to loss of block reward.

More specifically, $\mathsf{H}(\mathsf{B})_m$ should be set to the minimum non-negative number such that $\mathsf{P}^{(\mathsf{H}(\mathsf{B})_m+1)}(\mathsf{B})$ is correct on all the four fields of **deferredStateRoot**, **deferredReceiptsRoot**, **deferredLogsBloomHash**, and **blame**. For example, $\mathsf{H}(\mathsf{B})_m$ should be 1 if $\mathsf{P}(\mathsf{B})$ is incorrect in $\mathsf{H}(\mathsf{P}(\mathsf{B}))_r, \mathsf{H}(\mathsf{P}(\mathsf{B}))_e, \mathsf{H}(\mathsf{P}(\mathsf{B}))_b$ or $\mathsf{H}(\mathsf{P}(\mathsf{B}))_m$ while $\mathsf{P}^{(2)}(\mathsf{B})$ is correct in these fields.

Then, in the current block B's view, all blocks in between of B and $\mathsf{P}^{(\mathsf{H}(\mathsf{B})_m+1)}(\mathsf{B})$, i.e. $\mathsf{P}(\mathsf{B}), \cdots, \mathsf{P}^{(\mathsf{H}(\mathsf{B})_m)}(\mathsf{B})$, are committing to incorrect states, for which we say that those blocks are (directly) *blamed* by B. Furthermore, since B agrees with the state committed in $\mathsf{P}^{(\mathsf{H}(\mathsf{B})_m+1)}(\mathsf{B})$, the blocks blamed by $\mathsf{P}^{(\mathsf{H}(\mathsf{B})_m+1)}(\mathsf{B})$ is also (indirectly) blamed by B, and recursively we can determine all the blocks on $\mathsf{CHAIN}(\mathsf{B})$ that are blamed by the newest block B.

For blocks off $\mathsf{CHAIN}(\mathsf{B})$ (which is indeed the pivot chain in $\mathsf{PAST}(\mathsf{B})$ as long as B is valid), we do a best effort test to decide whether they are blamed by B. More precisely, every block B' must have determined a set of blocks blamed by B' along $\mathsf{CHAIN}(\mathsf{B}')$, and B' is blamed by B if they do not agree on exactly the same set of blamed blocks in the intersection $\mathsf{CHAIN}(\mathsf{B}) \bigcap \mathsf{CHAIN}(\mathsf{B}')$. We remark that no further check is applied on the execution related fields $\mathsf{B}'_r, \mathsf{B}'_e, \mathsf{B}'_b$ of B' as long as B' is off the current pivot chain.

Thus we have determined for every block B' whether it is blamed by any other block B. The punishment for blamed blocks (in particular, those blamed by the latest block on the pivot chain) is specified in Section 10.

In addition, the fields of state commitments, i.e. **deferredStateRoot**, **deferredReceiptsRoot**, **deferredLogsBloomHash**, would be handled differently when being blamed, as briefly described at the beginning of Section 3.4. That is, if $\mathsf{H}_m > 0$ and the those fields are blamed, then the commitment will be a Keccak hash of the vector consisting of the correct commitments. For example, if a block B has $\mathsf{H}(\mathsf{B})_m = 2$ such that $\mathsf{P}(\mathsf{B}), \mathsf{P}^{(2)}(\mathsf{B})$ are blamed but $\mathsf{P}^{(3)}(\mathsf{B})$ is not, then the **deferredStateRoot** field $\mathsf{H}(\mathsf{B})_r$ of block B would be

$$\mathsf{H}(\mathsf{B})_r = \mathsf{KEC}\left(\text{CorrectDeferredStateRoot}(\mathsf{B}), \text{CorrectDeferredStateRoot}\big(\mathsf{P}(\mathsf{B})\big), \text{CorrectDeferredStateRoot}(\mathsf{P}^{(2)}(\mathsf{B}))\right) \tag{52}$$

where CorrectDeferredStateRoot$(\cdot)$ is the function that returns the correct value of the deferred state root, which is a 256-bit Keccak hash, that should be filled in the given block. Note that 1) RLP serialization is not used in the vector since each element has fixed length, and 2) the vector length only depends on the **blame** field, regardless of correctness of individual fields in blamed blocks. Similar rules apply to **deferredReceiptsRoot** and **deferredLogsBloomHash**, i.e. $\mathsf{H}(\mathsf{B})_e$ and $\mathsf{H}(\mathsf{B})_b$.

4. Consensus

The consensus rules in Conflux are designed to make decision on two questions: the first is that whether a block is valid and should be added to the Conflux blockchain; the other is that in what order those valid blocks should be processed.

In an overview, the Conflux consensus protocol optimistically accepts all formally correct blocks and organizes them as a Tree-Graph (instead of a tree or a chain), and then specifies a total order of blocks in the Tree-Graph following the GHAST rule. This total order will be agreed by all honest participants and it is hard to change (under reasonable assumptions). After fixing such an order, transactions inside blocks are executed accordingly, and invalid transactions, which can be duplicating or conflicting previously processed transactions, are ignored.

4.1 Validation of Blocks

A set of Conflux blocks form a Tree-Graph structure where each vertex in the Tree-Graph represents a Conflux block and each directed edge in the Tree-Graph corresponds to a parent or referee reference. Every full node maintains a Tree-Graph structure of accepted blocks, which are blocks that are valid in the node's local view. Whenever receiving a new block, the full node verifies whether the block is valid before adding it into the Tree-Graph.

The validation of a new block can result in four outcomes:

- **suspending:** the block B is partially valid and $\mathsf{TimerDis}(\mathsf{B};\mathbf{G}) < 240$. ($\mathsf{TimerDis}(\cdot)$ is defined later in eq. (54) and $\mathbf{G}$ denotes the current Tree-Graph.) The status of block B will be checked again each time a new valid (or partially valid) block appears in the future set of B.
- **accept:** the block is valid (either fully valid or partially valid but not **suspending** anymore) and will be added to the Tree-Graph immediately.
- **reject:** the block is clearly invalid and will be discarded.
- **pending:** the block references some blocks not in the current Tree-Graph. Such blocks will be checked again once all the referenced blocks have been added to the Tree-Graph.

Given a new block B that is no longer **suspending**, the validation of B is done in the following steps:

1. **Header Validation.** This step asserts that B has a valid header (or at least a partially valid one) following Section 3.4.4 and 3.4.5. Note that a **Proof-of-Work Validation** of B is embedded inside the **Header Validation**, where the solution to the PoW puzzle is verified w.r.t. the legitimate target difficulty $\mathsf{H}(\mathsf{B})_d$. The canonical gas limit $\mathsf{H}(\mathsf{B})_\ell$ is also checked here, i.e. $\mathsf{H}(\mathsf{B})_\ell \in \left(1 \pm \frac{1}{1024}\right) \times \mathsf{H}(\mathsf{P}(\mathsf{B}))_\ell$ and $\mathsf{H}(\mathsf{B})_\ell \geq 5000$ as in (46).

   The **Proof-of-Work Validation** is the major mechanism against Sybil attacks and should be performed before invoking more expensive steps of verification and execution. It is interchangeable with other steps of the validation for better efficiency; and it will be performed last and repeatedly when mining a new block.

2. **Referee Validation.** The validation of referee headers asserts that B only references existing valid blocks. In case the new block references the head of an unknown block, it is marked as **pending** until all its referee blocks have been added to the Tree-Graph. The node is suggested (but not forced) to query its neighbors about the referenced unknown block.

3. **Volume Validtion.** The size of block body must not exceed 200 KB. Formally, $\sum_{\mathsf{T}\in\mathsf{B}_{\mathsf{Ts}}} |\mathsf{RLP}(\mathsf{T})| \leq 204800$.

4. **Internal Consistency.** This step asserts that B is self-consistent. More specifically:
   - the block header B_{H} is formally consistent with content in B_{Ts}, i.e. B_{H} is well-formed following Section 3.4.3;
   - every transaction $\mathsf{T} \in \mathsf{B}_{\mathsf{Ts}}$ is *locally legitimate*, which is the first test of the intrinsic validity of transactions:
     (a) T is well-formed RLP with no trailing bytes;
     (b) T has a valid signature by its sender $\mathsf{S}(\mathsf{T})$;
     (c) T has correct chain id $\mathsf{T}_c = 2$;
     (d) the **gasLimit** T_g is no smaller than the intrinsic gas g_0, where

$$
\begin{aligned}
g_0 \equiv & \sum_{i\in\mathsf{T_i},\mathsf{T_d}} \begin{cases} G_{\mathsf{txdatazero}} & \text{if } i = 0 \\ G_{\mathsf{txdatanonzero}} & \text{otherwise} \end{cases} \\
& + \begin{cases} G_{\mathsf{txcreate}} & \text{if } T_a = \varnothing \\ 0 & \text{otherwise} \end{cases} \\
& + G_{\mathsf{transaction}}
\end{aligned} \tag{53}
$$

   - the total gas consumption does not exceed the block gas limit, i.e. $\sum_{\mathsf{T}\in\mathsf{B}_{\mathsf{Ts}}} \mathsf{T}_g \leq \mathsf{B}_{\mathsf{H}_\ell}$.

5. If B passes all above steps, then it is marked **accept** and added into the Tree-Graph structure, otherwise it is marked **reject** and discarded.

Note that because a valid block must pass all the above validation steps and there is no jump or loop, all validation steps are interchangeable and parallelizable.

We emphasize an important difference of Conflux from other blockchains in validating blocks: in Conflux we check the validity of each transaction locally, i.e. at this moment we do not care if it is a duplicate of some processed transaction or the sender has insufficient balance. Thus a block B being valid **does not** imply that all transactions in B_{Ts} are valid or will be eventually executed. The validation of transactions in B_{Ts} will be deferred to the finalization of B.

4.2 Total Order in the Tree-Graph

Every Conflux full node maintains a Tree-Graph structure of accepted blocks, and now we discuss how to decide the total order of all accepted blocks.

Recall that in the Tree-Graph each vertex represents a Conflux block and each directed edge represents the reference of another block. The vertex for the genesis block has no outgoing edges, since the genesis block does not reference any other block. Other than the genesis block, each block has exactly one parent reference and possibly multiple (can be zero) referee references, represented by multiple outgoing edges from the corresponding vertex. This directed graph is acyclic since every directed edge reflects a clear chronological order of blocks, unless the referenced block is generated using a hash collision. Based on this Tree-Graph structure the Conflux consensus will first select a pivot chain that defines order of blocks on the chain, and then extend to the total order of all blocks.

4.2.1 GHAST and Weight Adaption on the Tree-Graph

In Conflux the valid blocks may have heterogeneous weights, even if there is no difficulty adjustment. This feature is introduced to improve the liveness guarantee in case a divergence of computing power (which is also the block generation power) happens, either by chance or caused by an active attack.

At a high level, the GHAST rule defines block weight as follows:

- In the usual case, all valid blocks have homogeneous weight. That is, every block with a proof of work satisfying the target difficulty of the epoch it belongs to would have the same weight, say weight 1.
- When a divergence of block generation power is observed, the distribution of block weights is changed adaptively:
  - A small fraction of blocks are marked as "heavy blocks" and given a greater weight. In particular, if the attached proof of work of a block has difficulty at least 250 times of the epoch's normal target difficulty, then this block is called a *heavy block* and its weight is also 250 times of a usual block, so as its base award.
  - Other blocks under this circumstance are still valid as long as the attached proof of work satisfies the normal target difficulty. But they have zero weight when making consensus decisions.
  - When the divergence is resolved, the weight distribution resumes to the usual case, where every block with a valid proof of work has the same weight as target difficulty of the epoch.

To be more specific, we remark that the GHAST rule only considers blocks in recent eras, e.g. the choice of parent edge is defined and validated *within* the subtree of the corresponding era genesis. See Section 4.3 for more details about era and checkpoints.

Timer Chain Conflux introduces the notion of timer block and timer chain as a robust estimation of elapsed time. Intuitively the timer chain is the longest chain that would be generated if the block generation were slow, or equivalently only high quality blocks were considered.

Every **fully valid** block B with quality $\mathsf{QUALITY}(\mathsf{B}) \geq 180 \cdot \mathsf{B}_d$ is called a *timer block*. The partial order of timer blocks can be deduced from the Tree-Graph of all blocks, which then determines the longest chain of timer blocks, called the *timer chain*. More specifically, the genesis block is also genesis of the timer chain, and the "parent timer block" of every timer block B is recursively defined by choosing the tip of timer chain in $\mathsf{PAST}(\mathsf{B})$. For convenience, the parent timer block of B is denoted by $\mathsf{P}_{timer}(\mathsf{B})$.

For convenience the timer chain in Tree-Graph $\mathbf{G}$ is denoted by $\mathsf{TimerChain}(\mathbf{G})$, and the difference of timer blocks in two Tree-Graphs $\mathbf{G}$ and $\mathbf{G}'$ is denoted by $\mathsf{TimerDis}(\mathbf{G}, \mathbf{G}')$ as follows

$$\mathsf{TimerDis}(\mathbf{G}, \mathbf{G}') \equiv \left|\mathsf{TimerChain}(\mathbf{G}) \backslash \mathbf{G}'\right| \tag{54}$$

The timer block difference $\mathsf{TimerDis}(\mathbf{G}, \mathbf{G}')$ provides an estimate of the time difference between two local views $\mathbf{G}$ and $\mathbf{G}'$.

Triggering Condition of Adaptive Weight Intuitively, a block B should apply the adaptive weight rule if any previous pivot block (on $\mathsf{CHAIN}(\mathsf{B})$) fails to receive support from a clear majority of computing power, i.e. the nemesis pivot block does not accumulate enough weight under its subtree after a sufficiently long time.

Formally, a block B is an *adaptive block*, denoted by $\mathrm{Adaptive}(\mathsf{B}) = \mathsf{True}$, if there exists $\mathsf{B}' \in \mathsf{CHAIN}(\mathsf{B})$ satisfying both of the following conditions:

1. $\mathsf{TimerDis}\left(\mathsf{PAST}(\mathsf{B}), \mathsf{PAST}\left(\mathsf{P}(\mathsf{B}')\right)\right) \geq 240$;
2. $\mathsf{SubTW}\left(\mathsf{B}'; \mathsf{PAST}(\mathsf{B})\right) - \left(\mathsf{SubTW}\left(\mathsf{P}(\mathsf{B}'); \mathsf{PAST}(\mathsf{B})\right) - \mathsf{SubTW}\left(\mathsf{B}'; \mathsf{PAST}(\mathsf{B})\right) - \mathsf{Weight}\left(\mathsf{P}(\mathsf{B}')\right)\right) < 1000 \cdot \mathsf{B}_d$,

   where $\mathsf{SubTW}(\mathsf{A}; \mathbf{G})$ denotes the total weight of the sub-tree rooted at A in graph $\mathbf{G}$, and $\mathsf{Weight}(\cdot)$ is the adaptive weight function as defined in (55).

Otherwise the block B is a *non-adaptive block* and denoted by $\mathrm{Adaptive}(\mathsf{B}) = \mathsf{False}$. The adaptive weight mechanism is only triggered for adaptive blocks.

Adaptive Block Weight Recalling that $\mathsf{PoW}(\cdot)$ denotes the proof-of-work function and B_d denotes the target difficulty of B, the GHAST rule formally defines the weight of a block B as follows:

$$\mathsf{Weight}(\mathsf{B}) \equiv \begin{cases} \mathsf{B}_d & \text{if } \neg\, \mathrm{Adaptive}(\mathsf{B}) \\ 0 & \text{if } \mathrm{Adaptive}(\mathsf{B}) \wedge 250 \cdot \mathsf{PoW}(\mathsf{B}_\mathsf{H}) \cdot \mathsf{B}_d > 2^{256} \text{ (i.e. } \mathsf{QUALITY}(\mathsf{B}) < 250 \cdot \mathsf{B}_d) \\ 250 \cdot \mathsf{B}_d & \text{if } \mathrm{Adaptive}(\mathsf{B}) \wedge 250 \cdot \mathsf{PoW}(\mathsf{B}_\mathsf{H}) \cdot \mathsf{B}_d \leq 2^{256} \text{ (i.e. } \mathsf{QUALITY}(\mathsf{B}) \geq 250 \cdot \mathsf{B}_d) \end{cases} \tag{55}$$

We remark that whether a block B should be adaptive or not is fully determined by $\mathsf{PAST}(\mathsf{B})$, which is indeed specified by B through directly and indirectly referenced blocks. Therefore it is not necessary to put a specific flag in the header of B for this condition, unless for efficiency concerns.

4.2.2 The GHAST Rule for Selecting the Pivot Chain

Since every block (except the genesis block) has exactly one parent, all parent edges in the Tree-Graph together form a *parental tree* with the genesis block being the root. In the parental tree, Conflux applies the GHAST rule to select a chain from the genesis block to one of the leaf blocks as the *pivot chain*, where blocks on the pivot chain are called *pivot blocks* and other blocks called *off-pivot blocks*.

In Conflux the pivot chain is not necessarily the longest chain or the "heaviest" chain. Indeed, the GHAST rule requires that the pivot chain should proceed to the branch whose subtree has greatest total weight (after weight adaption), in case there are multiple child blocks to choose. The Conflux selection algorithm starts from the genesis block. At each step, it computes the accumulated *total weight* of each child subtree in the parental tree and advances to the child block whose subtree has the largest total weight, until it reaches a leaf block in the local Tree-Graph.

The total weight of a subtree rooted at block B is denoted by B_t and defined recursively as:

$$\mathsf{B}_t \equiv \mathsf{B}_d + \sum_{\mathsf{B}':\mathsf{P}(\mathsf{B}')=\mathsf{B}} \mathsf{B}'_t \tag{56}$$

where $\mathsf{B}_d \equiv \mathsf{H}(\mathsf{B})_d$ denotes the target difficulty of B, and $\mathsf{P}(\mathsf{B}')$ is the parent block of B' (hence the summation is taken over B's children). Note that B_t is not a part of the block B – indeed it describes a state of B in the local view and may increase as more subsequent blocks are included afterwards.

We further remark that the total weight of every subtree can be computed from all block headers, since a block header contains the block difficulty as well as its parent and referees, which suffices to decide the weight of this block as in Section 4.2.1 and hence the total subtree weight recursively. As a result, the whole pivot chain can be determined by the block headers.

The advantage of the GHAST rule is that it guarantees the irreversibility of the selected pivot chain even if honest nodes fork because of network delays or other reasons. This is because forked blocks also contribute to the safety of the pivot chain (indeed, this property is inherited from GHOST rule, c.f. [1]).

For example, consider the local view as in Figure 1 and for simplicity suppose that all blocks have equal weight. Conflux would select Genesis, A, C, E, and H as pivot blocks. Note that they do not form the longest chain in the parental tree – the longest chain is Genesis, B, F, J, I, and K. Conflux does not select that longest chain because the subtree of A contains more blocks (and hence more total amount of computation) than the subtree of B. Therefore, the chain selection algorithm selects A over B at its first step.

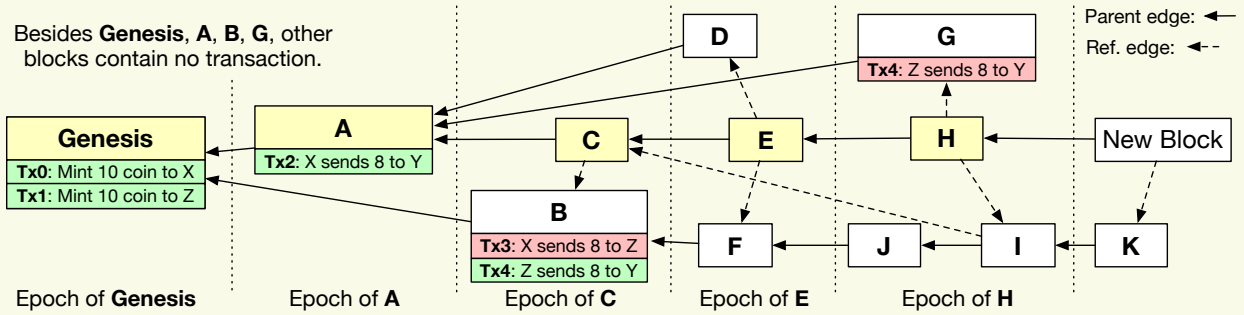


Figure 1. An example local Tree-Graph state to illustrate the consensus algorithm of Conflux. The yellow blocks are on the pivot chain in the Tree-Graph. Each block on the pivot chain forms a new epoch to partition blocks in the Tree-Graph.

4.2.3 Epoch

Given the pivot chain in a Tree-Graph, Conflux splits all blocks into epochs as follows:

- Every pivot block B is at epoch $\mathsf{H}(\mathsf{B})_h$, or simply the *epoch of* B, which is denoted by $\mathsf{EPOCH}(\mathsf{B})$. In particular the genesis block is at epoch 0.
- Every off-pivot block B' is at the epoch of the first pivot block B that references it, directly or indirectly. That is, every block $\mathsf{B}' \in \mathsf{U}(\mathsf{B})$ is at the epoch of B; then every block B'' referenced by B', i.e. $\mathsf{B}'' \in \mathsf{U}(\mathsf{B}') \cup \{\mathsf{P}(\mathsf{B}')\}$, if not already included in an earlier epoch, is also at the epoch of B; and recursively all blocks referenced by B'' and so on.

In other words, the epoch of B consists of all blocks within the local view of B, such that there blocks are potentially produced after $\mathsf{P}(\mathsf{B})$ but clearly before B. For example, each of the pivot blocks Genesis, A, C, E, and H corresponds to one individual epoch in Figure 1. The block J belongs to the epoch of H but not the epoch of E, because J is reachable from H but not reachable from the previous pivot block E.

It is clear from the definition that as long as the pivot chain is not reverted, the partition of epochs cannot be changed.

Epoch capacity Each epoch now has a limit of executing 200 blocks. When a single epoch contains more than 200 blocks, Conflux will only execute the last 200 blocks (as specified in Section 4.2.4) in that epoch. This limit is introduced to battle DoS attacks about releasing a lot of blocks suddenly.

4.2.4 Total Order of Blocks

Conflux extends the total order of pivot blocks to all blocks in a Tree-Graph as follows. Conflux first sorts blocks according to their corresponding epochs, so that a block in an earlier epoch always precedes another block in a later epoch; and then Conflux sorts the blocks inside each epoch based on their topological order, i.e. corresponding to the partial order implied by referee references. In case two blocks have no partial order relation, Conflux breaks ties deterministically with the unique ids of these two blocks. More detailed rules are described with codes in Figure 2.

Input : A block B and the local Tree-Graph $\mathbf{G}$ (with B in $\mathbf{G}$ and G is the genesis block of $\mathbf{G}$)
Output : A list of blocks $\mathbf{L} = \mathsf{B}_1 \circ \mathsf{B}_2 \circ \ldots \circ \mathsf{B}_n$, where $\mathsf{B}_1, \ldots, \mathsf{B}_n$ are blocks in $\mathbf{G}$, and in particular $\mathsf{B}_1 = \mathsf{G}$ and $\mathsf{B}_n = \mathsf{B}$.
A list of pivot blocks $\mathbf{P} = \mathsf{B}'_1 \circ \mathsf{B}'_2 \circ \cdots \circ \mathsf{B}'_m$ where $\mathsf{B}'_1, \cdots, \mathsf{B}'_m$ are pivot blocks and in particular $\mathsf{B}'_1 = \mathsf{G}$ and $\mathsf{B}'_m = \mathsf{B}$.

$\mathsf{B}' \longleftarrow \mathsf{P}(\mathsf{B})$
if $\mathsf{B}' = \perp$ **then**
  return (B,B)
$(\mathbf{L},\mathbf{P}) \longleftarrow \text{ConfluxOrder}(\mathsf{B}';\mathbf{G})$
$\mathbf{L}' \longleftarrow$ An empty list
$\Delta \longleftarrow \mathsf{PAST}(\mathsf{B}) \setminus (\mathsf{PAST}(\mathsf{B}') \cup \{\mathsf{B}'\})$
while $\Delta \neq \varnothing$ **do**
  $\Delta' \longleftarrow \Delta \setminus (\cup_{\mathsf{A} \in \Delta} \mathsf{PAST}(\mathsf{A}))$
  $\mathsf{B}'' \longleftarrow \arg\max_{\mathsf{A} \in \Delta'} \{\mathsf{Hash}(\mathsf{A})\}$
  $\mathbf{L}' \longleftarrow \mathsf{B}'' \circ \mathbf{L}'$
  $\Delta \longleftarrow \Delta \setminus \{\mathsf{B}''\}$
$\mathbf{L} \longleftarrow \mathbf{L} \circ \mathbf{L}' \circ \mathsf{B}$
$\mathbf{P} \longleftarrow \mathbf{P} \circ \mathsf{B}$
return $(\mathbf{L},\mathbf{P})$

Figure 2. The Definition of the ConfluxOrder function.

For example, the local Tree-Graph in Figure 1 may give a total order as Genesis, A, B, C, D, F, E, G, J, I, H, and K. The order of D and F may change if the block id of F is smaller than D, and the same holds for G, J, and I.

Here we remark that the weight of blocks is not used when extending the pivot chain to the total order of all blocks. All blocks, even blocks with zero weight after adaption, are treated equally in this procedure.

4.2.5 Total Order of Transactions

Conflux first sorts transactions based on the relative order of their enclosing blocks. In case two transactions belong to the same block, Conflux sorts them based on their appearance order in that block. Conflux checks conflicts of transactions at the same time when deriving the orders. If two transactions are conflicting with each other, e.g. they have exactly the same sender and nonce, then Conflux will attempt to execute the first one and ignore the second one if the first one is executed. If one transaction appears in multiple blocks, Conflux will keep the first valid appearance of this transaction and discard all redundant ones.

For example, the transaction total order in Figure 1 is Tx0, Tx1, Tx2, ~~Tx3~~, Tx4, and ~~Tx4~~, where Conflux discards Tx3 because it conflicts with a previously executed valid transaction Tx2, and discards the second appearance of Tx4 because it is redundant.

4.3 Checkpoint

Conflux implements a checkpoint mechanism to tailor the ever-growing blockchain history data. At a high level, every 50000 epochs form an *era*[3] and a *checkpoint* is made when the era genesis block becomes stable. Thereafter, full nodes can safely discard the content of old blocks.

4.3.1 Era and Era Genesis

Formally, eras in Conflux are partitioned with respect to the height of pivot blocks. Every 50000 height corresponds to one era, and in particular the pivot block of height $50000 \cdot N$ is called the *era genesis block of Era N*. For example, the genesis block, which is the only block at height 0, is the era genesis block of Era 0 (i.e. the first era); the pivot block at height 50000 is the era genesis block of Era 1 (i.e. the second era). Furthermore, an era genesis will eventually be finalized and become a *stable era genesis*, which is irreversible in any case.

The stable era geneses are recursively defined as below:

1. The genesis block is the first stable era genesis.
2. An era genesis block G becomes a stable era genesis if it satisfies the following:
   - G is in the subtree of the last stable era genesis;
   - the subtree of G contains 240 *consecutive* timer blocks ended by B, i.e. $\exists \mathsf{B}$ such that $\mathsf{B}, \mathsf{P}_{timer}(\mathsf{B}), \ldots, \mathsf{P}^{(239)}_{timer}(\mathsf{B})$ are all in the subtree of G;
   - there are 240 *consecutive* timer blocks in the future of B (these blocks are not necessarily within the subtree of G).
3. Era genesis blocks preceding a stable era genesis are also stable.

Let the latest stable era genesis block of a Tree-Graph $\mathbf{G}$ be denoted by $\mathsf{StableEraGenesis}(\mathbf{G})$. The GHAST rule, including weight adaption and parent selection (as specified in Section 4.2.1 and 4.2.2), are applied within the subtree of $\mathsf{StableEraGenesis}(\mathbf{G})$. The anti-cone penalty, as defined in Section 10.1.1, is also subject to blocks within the subtree of $\mathsf{StableEraGenesis}(\mathbf{G})$, though such restriction makes no essential difference according to current anti-cone parameters.

4.3.2 Truncation of Consensus Tree-Graph

In a Tree-Graph $\mathbf{G}$, the latest stable era genesis $\mathsf{StableEraGenesis}(\mathbf{G})$ is indeed a checkpoint that can be treated as the "new genesis block" thereafter, since the consensus rules only apply to blocks within the subtree of $\mathsf{StableEraGenesis}(\mathbf{G})$. Therefore, the bodies of blocks in $\mathsf{PAST}(\mathsf{StableEraGenesis}(\mathbf{G}))$ are no longer needed for future world-states and can be safely discarded from every full node's memory. However, the headers of those blocks in $\mathsf{PAST}(\mathsf{StableEraGenesis}(\mathbf{G}))$ are still needed for validation of references from future blocks (blocks outside the future set $\mathsf{FUTURE}(\mathsf{StableEraGenesis}(\mathbf{G}); \mathbf{G})$ may reference blocks in $\mathsf{PAST}(\mathsf{StableEraGenesis}(\mathbf{G}))$).

For convenience we let B' and B'' denote the latest two stable era geneses as follows:

$$\mathsf{B}' \equiv \mathsf{StableEraGenesis}(\mathbf{G})$$
$$\mathsf{B}'' \equiv \mathsf{StableEraGenesis}(\mathsf{PAST}(\mathsf{B}'))$$

Blocks missing one stable era genesis, i.e. the blocks in $\mathsf{FUTURE}(\mathsf{B}''; \mathbf{G}) \backslash \mathsf{FUTURE}(\mathsf{B}'; \mathbf{G})$, can be referenced by a new block in $\mathbf{G}$ but will not affect consensus decision. Blocks missing two stable era geneses, i.e. the blocks outside

[3] In geologic time scale, *eons* are divided into *eras*, which are in turn divided into *periods*, *epochs* and *ages*.

$\mathsf{FUTURE}(\mathsf{B}'';\mathbf{G})$, cannot be referenced. More specifically, when adding a new block B into the Tree-Graph $\mathbf{G}$, the validity of B is examined as follows:

1. If $\mathsf{B} \notin \mathsf{FUTURE}(\mathsf{B}'';\mathbf{G})$, then B is discarded.
2. If $\mathsf{B} \in \mathsf{FUTURE}(\mathsf{B}'';\mathbf{G}) \backslash \mathsf{FUTURE}(\mathsf{B}';\mathbf{G})$, then B is partially valid. That is, the block B has zero weight in consensus decision and should not be referenced, but the transactions in B_{Ts} are still valid.
3. If $\mathsf{B} \in \mathsf{FUTURE}(\mathsf{B}';\mathbf{G})$ but $\mathsf{P}(\mathsf{B}) \notin \mathsf{SubTree}(\mathsf{B}'';\mathbf{G})$, then B is also partially valid.
4. Otherwise follow the rules in Section 3.4.5.

4.4 Finalization

In this part we discuss when a block or a transaction in Conflux is considered irreversible, or "finalized". This is crucial for off-chain users to decide when to confirm a transaction or a state.

Like every other PoW consensus system, the risk that an adversary succeeds in reverting the blockchain history decreases over time but never goes to zero. That is, there is always a positive probability, no matter how tiny it is, that an adversary generates a branch with higher accumulated weight than the one generated by honest parties. This is a nature of PoW consensus but not one weakness, because: 1) a sufficiently small probability is usually considered equivalent to zero in practice; 2) that tiny probability can be made even smaller than the probability that an adversary breaks the public-key cryptosystem or finds a collision of the hash reference, and hence not the bottleneck of security. Therefore, in order to decide the concrete finalization rule, a user must specify how much risk he can tolerate as well as his (perhaps subjective) assumption about several system parameters.

In what follows we elaborate the finalization of a block. A transaction T is finalized if the first block B' where T is **executed** becomes finalized. This is a sufficient but sometimes not necessary condition[4]. Note that "the first block including T becomes finalized" is insufficient, since Conflux allows invalid transactions.

Consider the block B' in a Tree-Graph and suppose that B' belongs to the epoch of a pivot block B. Then the finalization of B' reduces to the finalization of B on the pivot chain. The user can decide whether B is finalized as follows:

1. Estimate or make assumptions about following system parameters:
   - **Block Generation Rate:** Let $q \equiv \lambda_a/\lambda_h$ where λ_h denotes the combined block generation rate of honest nodes and λ_a denotes the block generation rate of attacker. The user needs to make an assumption of the attacker's power by setting an upper bound for q.
   - **Network Synchronization:** If at time t, an honest node broadcasts a block via the gossip network, then by time $t+d$ all honest nodes receive this block (the nodes not receiving by time $t+d$ will be counted as adversary). According to our experiment in a globally distributed testing environment, we set d equal to 10 seconds.
2. Make sure that all the pivot blocks preceding B have been finalized.
3. Make sure that block B stays on the pivot chain since $2d$ time ago.
4. For $q \leq 0.2$, compute the confirmation risk r_1 according to the **Fast Confirmation Rule**.
   - The fast confirmation rule assumes that the GHAST weight adaption is not triggered under the subtree of $\mathsf{P}(\mathsf{B})$ during the generation of 8000 blocks (≈ 1.1 hours) since the creation of $\mathsf{P}(\mathsf{B})$.
   - Let r_2 denote the risk that an attacker breaks the above assumption.
5. For $q < 1$, compute the confirmation risk r_3 according to the **Slow Confirmation Rule**.
6. The confirmation risk for B is bounded as $\mathsf{Risk}(\mathsf{B}) \leq \min\{r_1 + r_2, r_3\}$.

4.4.1 Fast/Slow Confirmation Rules

Since evaluating $\mathsf{Risk}(\mathsf{B})$ may be costly, we provide the following simple (but not tight) formulas for risk estimation. These estimations are conservative and hence result in a longer confirmation time. Users, especially those who are also developers, are encouraged to make more accurate estimation and design sophisticated strategies to achieve a better balance between confirmation time and security.

For the estimation of confirmation risk, we introduce the following values with respect to the current local Tree-Graph $\mathbf{G}$ and any block $\mathsf{A} \in \mathbf{G}$.

[4] The finalization of a transaction may be much faster. For example, a simple transaction T may be safely confirmed before the blocks containing T being finalized, if the execution of T is indifferent of the agreement of pivot chain, i.e. no conflicting or dependent transactions of T included in competing blocks.

- c_0: the total weight of all blocks locally observable in $\mathbf{G}$;
- c_1: the total weight of received blocks in the past $2d$ time (in local clock);
- $\mathbf{d}$: the largest target difficulty in the last one day (in local clock);
- $w_1(\mathsf{A})$: the total weight of blocks under the subtree rooted at A in $\mathbf{G}$, i.e. $w_1(\mathsf{A}) \equiv \sum_{\mathsf{A}' \in \mathbf{G}:\mathsf{A} \in \mathsf{CHAIN}(\mathsf{A}')} \mathsf{Weight}(\mathsf{A}')$;
- $w_2(\mathsf{A})$: the maximum weight of subtrees rooted at blocks in $\mathsf{SIBLING}(\mathsf{A})$, i.e. $w_2(\mathsf{A}) \equiv \max_{\mathsf{A}' \in \mathsf{SIBLING}(\mathsf{A})} \{w_1(\mathsf{A}')\}$ (here malicious blocks can be excluded for better performance, while failing to exclude malicious blocks would result in a correct but more conservative (less accurate) estimation for the upper bound of confirmation risk);
- $w_3(\mathsf{A})$: the total weight of all the blocks in $\mathsf{PAST}(\mathsf{A})$, i.e. $w_3(\mathsf{A}) \equiv \sum_{\mathsf{A}' \in \mathsf{PAST}(\mathsf{A})} \mathsf{Weight}(\mathsf{A}')$.

Then we let n be the estimation of B's advantage over its siblings, and m denote the total (equivalent) number of blocks competing with B, where n and m are formally defined as below:

$$n \equiv \lceil (w_1(\mathsf{B}) - w_2(\mathsf{B}) - c_1)/\mathbf{d} \rceil, \qquad m \equiv \lfloor (c_0 - w_3(\mathsf{P}(\mathsf{B})))/\mathbf{d} \rfloor \tag{57}$$

If n approaches m quickly then it must be the case that most of the mining power are concentrating under the subtree of B, in which case the block B can be finalized using the Fast Confirmation Rule. However, it is possible that the Fast Confirmation Rule is never satisfied for a predetermined risk tolerance in case the convergence of mining power is not that timely. Then the confirmation of B has to rely on the Slow Confirmation Rule, which takes time but will eventually come true as long as the majority of mining power is honest.

Table 1. The confirmation risk lookup table for Fast Confirmation Rule.

Risk tolerance \ Adversary power	$q \leq 0.1$ (i.e. 9.1% adversary)	$q \leq 0.2$ (i.e. 16.7% adversary)
$\mathsf{Risk}(\mathsf{B}) < 10^{-4}$	$m - n \leq \min\{0.85m - 12, 4400\}^*$	$m - n \leq \min\{0.75m - 22, 2250\}$
$\mathsf{Risk}(\mathsf{B}) < 10^{-6}$	$m - n \leq \min\{0.80m - 12, 3800\}$	$m - n \leq \min\{0.70m - 22, 1500\}$
$\mathsf{Risk}(\mathsf{B}) < 10^{-8}$	$m - n \leq \min\{0.75m - 12, 3200\}$	$m - n \leq \min\{0.65m - 22, 750\}$

* Slightly better security than confirmation with 6 successive blocks in Bitcoin.

Table 2. The confirmation risk lookup table for Slow Confirmation Rule.

Risk tolerance \ Adversary power	$q \leq 0.25$ (i.e. 20% adversary)	$q \leq 0.5$ (i.e. 33.3% adversary)
$\mathsf{Risk}(\mathsf{B}) < 10^{-4}$	$m - n \leq 0.6m - 5700^\dagger$	$m - n \leq 0.35m - 10900$
$\mathsf{Risk}(\mathsf{B}) < 10^{-6}$	$m - n \leq 0.6m - 7200$	$m - n \leq 0.35m - 13600$
$\mathsf{Risk}(\mathsf{B}) < 10^{-8}$	$m - n \leq 0.6m - 8700$	$m - n \leq 0.35m - 16200$

† Security equivalent to confirmation with $13 \sim 14$ successive blocks in Bitcoin.

Safety of Fast Confirmation Rule. Now we upper bound the risk r_2 that the GHAST weight adaption is triggered within 8000 successive blocks since the generation of $\mathsf{P}(\mathsf{B})$.

Let h be the height of the latest stable era genesis block in the current Tree-Graph $\mathbf{G}$. Let B_i denote the block at height $h+i$ on $\mathsf{CHAIN}(\mathsf{B})$. Let ψ be an arbitrary positive integer (e.g. $\psi = 50$). For integer $j \in \mathbb{N}$, let m' and m_j, n_j be defined as follows:

$$m' \equiv \lfloor (c_0 - w_3(\mathsf{P}(\mathsf{B})))/\mathbf{d} \rfloor \tag{58}$$

$$n_j \equiv \lceil (w_1(\mathsf{B}_{(j+1)\cdot\psi}) - \max_{i \in (j\cdot\psi, (j+1)\cdot\psi]} w_2(\mathsf{B}_i) - c_1)/\mathbf{d} \rceil - m' \tag{59}$$

$$m_j \equiv \lfloor (c_0 - w_3(\mathsf{B}_{j\cdot\psi}))/\mathbf{d} \rfloor \tag{60}$$

Let j_{max} be the largest integer satisfying $\mathsf{TimerDis}(\mathbf{G}, \mathsf{PAST}(\mathsf{B}_{j_{max}\cdot\psi})) \geq 140$. For $q \leq 0.2$, the risk r_2 is bounded by

$$10^{-7} + \sum_{j=0}^{j_{max}} 10^{(m_j/3 - n_j)/700 + 5.3} \tag{61}$$

5. Blockchain Execution

After determining the total order of blocks, the transactions are executed as if they are packed into sequential blocks on an Ethereum-like chain.

Blockchain execution is based on a series of ordered blocks $\mathbf{L}$ and a subsequence of pivot blocks $\mathbf{P}$ output by figure 2. The pivot blocks divided into $\mathbf{L}$ into several epochs. For $k \geq 1$, the epoch k (denoted by $\mathbf{E}_k$) refers the slice in $\mathbf{L}$ started with the next block of $\mathbf{P}[k-1]$ and ended at block $\mathbf{P}[k]$. The epoch 0 refers the genesis block.

5.1 Initial state

The initialization world state σ^0 is set as follows. A list $\mathbf{a}$ with elements (a,b) gives the addresses a and their balance b when the Conflux blockchain launched.

$$\forall (a,b) \in \mathbf{a}, \sigma^0[a] = \alpha^0 \quad \text{except:} \alpha_b = b \tag{62}$$

The global statistic information will be set as follows:

$$\sigma^0[a_{\mathsf{stake}}][k_1]_v \equiv 63072000 \times 2^{80} \tag{63}$$
$$\sigma^0[a_{\mathsf{stake}}][k_2]_v \equiv 63072000 \times 40000 \tag{64}$$
$$\sigma^0[a_{\mathsf{stake}}][k_3]_v \equiv 0 \tag{65}$$
$$\sigma^0[a_{\mathsf{stake}}][k_4]_v \equiv 0 \tag{66}$$
$$\sigma^0[a_{\mathsf{stake}}][k_5]_v \equiv \sum_{(a,b)\in\mathbf{a}} b \tag{67}$$
$$\text{where:} \tag{68}$$
$$a_{\mathsf{stake}} \equiv 0\text{x}0888000000000000000000000000000000000002 \tag{69}$$
$$k_1 \equiv [\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{70}$$
$$k_2 \equiv [\mathsf{interest\_rate}]_{\mathsf{ch}} \tag{71}$$
$$k_3 \equiv [\mathsf{total\_staking\_tokens}]_{\mathsf{ch}} \tag{72}$$
$$k_4 \equiv [\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{73}$$
$$k_5 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{74}$$

5.2 Epoch execution

The blockchain is executed epoch by epoch started with epoch 1. Let σ_{k-1} denote the world state after the execution of epoch $k-1$. The Conflux protocol updates world state from σ_{k-1} to σ_k as follows. Besides updating the world state, the protocol also generates a receipt list $\mathbf{R}_k$ for epoch execution.

Blocks execution. First, all the blocks in epoch are executed in sequence by block execution function $\mathscr{C}_{\mathsf{block}}(\sigma, \mathsf{B}, \mathbf{L}[0..(\tau - 1)]) = (\sigma', \mathbf{R}')$, where B is the block to be executed, τ is the the index of block B in $\mathbf{L}$ and $\mathbf{R}'$ is the sequence of transaction receipts. After executing all the blocks, the resultant world-state σ^* becomes the input of the next step and the concatenation of block receipts becomes epoch receipts $\mathbf{R}$. Function $\mathscr{C}_{\mathsf{block}}(\sigma, \mathsf{B}, \mathbf{L})$ is defined in section 5.3.

Distribute mining reward. Since Conflux incentive mechanism puts off the mining reward distribution for 12 epochs, after execution of epoch k, Conflux distribute the mining reward for blocks in $\mathbf{E}_{k-12}$. The computing of mining reward for blocks in epoch $k-12$ requires the following context information.

- The epoch block set $\mathbf{E}_{k-12}$
- The world-state before the execution of all the block B in $\mathbf{E}_{k-12}$, denoted by $\sigma(\mathsf{B})$.
- The transaction receipts of all the block B in $\mathbf{E}_{k-12}$, denoted by $\mathbf{R}'(\mathsf{B})$.
- The tree-graph structure for blocks in $\mathsf{PAST}(\mathbf{P}[k-12+10])$.

Section 10 describes how to compute the block reward $\mathsf{R}(\mathsf{B})$ with the context information. The mining reward will be distributed to the block author if the author is . The global parameter *total issued tokens* is updated accordingly. Suppose σ^* is the world state after blocks execution, it will be updated to σ^{**} by

$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{75}$$
$$\forall a \in \mathbb{B}_{160} \text{ with } \mathsf{Type}_a \in \{[0000]_2, [0001]_2, [1000]_2\} \tag{76}$$
$$\sigma^{**}[a]_b \equiv \sigma^*[a]_b + \sum_{\mathsf{B}\in\mathbf{E}_{k-12}} \mathbb{I}(\mathsf{B}_{\mathsf{H}_a} = a) \times \mathsf{R}(\mathsf{B}) \tag{77}$$
$$\sigma^{**}[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] \equiv \sigma^*[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] + \sum_{\mathsf{B}\in\mathbf{E}_{k-12}} \left(\mathbb{I}(\mathsf{Type}_{\mathsf{B}_{\mathsf{H}_a}} \in \{[0000]_2, [0001]_2, [1000]_2\}) \times \mathsf{R}(\mathsf{B}) - \sum_{R\in\mathbf{R}'(\mathsf{B})} R_f \right) \tag{78}$$
$$\text{where:} \tag{79}$$
$$a_{\mathsf{stake}} \equiv 0\text{x}0888000000000000000000000000000000000002 \tag{80}$$
$$k_3 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{81}$$

5.3 Block execution

The block execution function $\mathscr{C}_{\mathsf{block}}(\sigma, \mathsf{B}, \mathbf{L})$ consists of two steps.

Update accumulate interest

$$\sigma^* \equiv \sigma \qquad \text{except:} \tag{82}$$

$$\sigma^*[a_{\mathsf{stake}}]_{\mathbf{s}}[k_1] \equiv \left\lfloor \sigma[a_{\mathsf{stake}}]_{\mathbf{s}}[k_1] \times \left(1 + \frac{4\%}{63072000}\right) \right\rfloor \tag{83}$$

$$\text{where:} \tag{84}$$

$$a_{\mathsf{stake}} \equiv 0\text{x}0888000000000000000000000000000000000002 \tag{85}$$

$$k_1 \equiv [\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{86}$$

Execute transactions in block Each block B contains a series of transactions B_{Ts}. Start with world state σ^*, Conflux executes these transactions in sequence by a transform function $\Upsilon(\sigma, \mathsf{T}, \mathbf{L}) = (\sigma', R)$, which updates world state from σ to σ' by processing transaction T and outputs a receipt R. The input $\mathbf{L}$ represents the blocks in front of the present block.After executing all the transaction, the resultant state σ' and the concatenation of all the transaction receipts $\mathbf{R}'$ consists of the output of function $\mathscr{C}_{\mathsf{block}}$.

6. Transaction Processing

Conflux implements the same virtual machine as Ethereum [3]. The execution of a transaction defines the transform function $\Upsilon(\sigma, \mathsf{T}, \mathbf{L})$, which is similar with Ethereum's state transition function.

6.1 Overview

In what follows we focus on the Conflux specific designs in the execution.

Gas and Payment As defined in Section 3.3 every transaction T has two fields of **gasLimit** and **gasPrice** that declare the specific amount of associated gas T_g and the price T_p of per unit gas. When starting the execution of a transaction T, the purchase of gas happens at the price $\mathsf{T}_g \times \mathsf{T}_p$ and the transaction T is considered invalid if the actor responsible for the cost of gas consumption cannot afford such a purchase. Normally $\mathsf{S}(\mathsf{T})$, the sender of T, is responsible for the cost of gas consumption. But the gas consumption maybe sponsored by called contract sometimes. (See the following paragraph for details.)

Like in Ethereum, gas does not exist outside the execution of transactions.

The unused gas can be refunded after the transaction T is executed, but no more than a quarter of the total value spent on purchasing. Thus, the *refundable amount of gas* $g^\dagger$ is the minimum of the *legitimately remaining gas* g' and a quarter of the **gasLimit** of T, i.e. $g^\dagger \equiv \min\{g', \mathsf{T}_g/4\}$, where in principle no gas is refundable (i.e. $g^\dagger = g' = 0$) if the execution of T fails due to the sender's fault.

The actor who initially purchased the gas for T will get the refund of $g^\dagger \times \mathsf{T}_p$. And the CFX paid for the consumed gas is

$$(\mathsf{T}_g - g^\dagger) \times \mathsf{T}_p. \tag{87}$$

If the T is not executed (i.e. only when $R_z = 2$ as in Section 6.2.1)), no gas will be charged. If the sender $\mathsf{S}(\mathsf{T})$ can not afford gas fee, all its the remained balance of sender $\mathsf{S}(\mathsf{T})$ will be charged as gas fee. The actual charged gas fee will be record in receipt R_f.

The charged gas fee is added to the reward pool for miners. Thus in general a higher gas price on a transaction would cost the sender more but also increase the chance of being processed timely.

In computing the accumulated gas used of the whole block, the non-refundable gas $g^\dagger - g'$ is not taken into consideration. But in case the sender $\mathsf{S}(\mathsf{T})$ can not afford gas fee, the gas used is considered as T_g, even if the actual charged gas fee is less than $\mathsf{T}_p \times \mathsf{T}_g$. The gas used is also recorded in receipt R_g. Thus the accumulated gas used of a block is intrinsically taking the summation of field R_g over transaction receipts.

Storage Limit and Storage Collateral Every transactions also have a filed of **storageLimit** that declare the maximum storage bytes T_ℓ increasing for the present transaction. Before transaction execution, besides gas fee and transferred value, the sender $\mathsf{S}(\mathsf{T})$ must have enough balance for storage collateral for specified **storageLimit**, i.e., $\mathsf{T}_\ell \times \frac{10^{18}}{1024}$ Drip. Unlike the gas fee, these collateral will not be charged or locked at this time. At the end of transaction execution, if the sender doesn't have enough balance paying for the increased storage collateral or the increased storage bytes of sender exceeds storage limit, the transaction execution fails. More details for collateral for storage is specified in section 7.

Sponsorship In case the transaction T is calling a smart contract $\mathbf{C}$ with **sponsor for gas** and T is qualified for the subsidy as checked in function $\mathsf{Whitelist}(\cdot)$ defined in eq. (261), $\mathbf{C}$ is responsible for the purchase of gas if it has sufficient **sponsor balance for gas** and the transaction **gasLimit** T_g does not exceed the **sponsor limit for gas**, and otherwise the sender $\mathsf{S}(\mathsf{T})$ is still responsible for the whole purchase of gas as if there were no sponsor at all. Formally, we define function $\mathsf{GasElig}(\sigma,\mathsf{T})$ to check whether transaction T is eligible for gas consumption sponsorship and define function $\mathsf{GasSpr}(\sigma,\mathsf{T})$ to check where T is actually sponsored for gas consumption.

$$\mathsf{GasElig}(\sigma,\mathsf{T}) \equiv \quad \mathsf{Type}_{\mathsf{T}_a} = [1000]_2 \wedge \mathsf{Whitelist}(\sigma,\mathsf{S}(\mathsf{T}),\mathbf{C}) \wedge \sigma[\mathsf{T}_a]_p[\mathsf{gas}]_a \neq 0 \wedge \mathsf{T}_g \times \mathsf{T}_p \leq \sigma[\mathsf{T}_a]_p[\mathsf{limit}] \tag{88}$$

$$\mathsf{GasSpr}(\sigma,\mathsf{T}) \equiv \quad \mathsf{GasElig}(\sigma,\mathsf{T}) \wedge \sigma[\mathsf{T}_a]_p[\mathsf{gas}]_b \geq \mathsf{T}_g \times \mathsf{T}_p \tag{89}$$

where function $\mathsf{Whitelist}(\cdot)$ is defined in eq. (261).

If a contract has **sponsor for collateral**, the storage collateral can also be sposnored by a contract. Formally,

$$\mathsf{ColElig}(\sigma,\mathsf{T}) \equiv \quad \mathsf{Type}_{\mathsf{T}_a} = [1000]_2 \wedge \mathsf{Whitelist}(\sigma,\mathsf{S}(\mathsf{T}),\mathbf{C}) \wedge \sigma[\mathsf{T}_a]_p[\mathsf{col}]_a \neq 0 \tag{90}$$

$$\mathsf{ColSpr}(\sigma,\mathsf{T}) \equiv \quad \mathsf{ColElig}(\sigma,\mathsf{T}) \wedge \sigma[\mathsf{T}_a]_p[\mathsf{col}]_b \geq \mathsf{T}_\ell \times \frac{10^{18}}{1024} \tag{91}$$

6.2 Transaction Execution

6.2.1 Pre-execution Validation

Before being executed, a transaction T in the processing queue must pass the following secondary test of intrinsic validity.

1. The current epoch is in the range specified by **epochHeight**, i.e. current epoch height is in $[\mathsf{T}_e - 100000, \mathsf{T}_e + 100000]$.
2. The transaction **nonce** is valid, i.e. $\mathsf{T}_n = \sigma[\mathsf{S}(\mathsf{T})]_n$ where σ is the current world-state.
3. The recipient address is valid , i.e. the type indicator (first 4-bit) of T_a belongs to $\{[0000]_2, [0001]_2, [1000]_2\}$.

Note that the local legality of the transaction, e.g. the RLP format and the validity of signature, is already verified in the first intrinsic validity test before accepting the corresponding block into the Conflux Tree-Graph, as discussed in Section 4.1, and will not be checked again at this moment.

If T fails at these checks, the transaction will not be executed, the nonce for account will not increase and no transaction fee is charged for such transaction. Let R' be the receipt of last transaction. Then the receipt of current transaction will be set as follows:

$$R_u = R'_u \qquad R_f = 0 \qquad R_g = 0 \qquad R_{\mathbf{l}} = \varepsilon \qquad R_z = 2 \qquad R_s = 0 \qquad R_{\mathbf{o}} = \varepsilon \qquad R_{\mathbf{i}} = \varepsilon \tag{92}$$

(The bloom filter R_b of log $R_{\mathbf{l}}$ is computed accordingly.)

If T passes all the above pre-execution checks, the execution of T is as specified in the rest of this section.

6.2.2 Preprocessing

In the preprocessing phase of T, the balance of $\mathsf{S}(\mathsf{T})$ (and the sponsor, if applicable) is examined so that the payment for any further operation is assured. The world-state will be transformed from σ into $\sigma^0 \equiv \sigma^{**}$ if T passes the preprocessing, or directly into σ' and the execution is aborted if T fails at any step.

Nonce incremental. The beginning of execution causes an irrevocable changed to the state σ: the nonce of the sender, $\mathsf{S}(\mathsf{T})_n$, is incremented by one. We define the state σ^*:

$$\sigma^* \equiv \sigma \qquad \text{except:} \tag{93}$$

$$\sigma^*[\mathsf{S}(\mathsf{T})]_n \equiv \sigma[\mathsf{S}(\mathsf{T})]_n + 1 \tag{94}$$

Gas consumption payment validation. The up-front payment of a transaction T first figures out whether the gas consumption is sponsored. T is sponsored on gas consumption if T is eligible for sponsorship on gas consumption and the calling contract has sufficient **sponsor balance for gas fee**.

- If the gas consumption of T is sponsored, the world-state σ^{**} after gas consumption payment is as follows:

$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{95}$$

$$\sigma^{**}[\mathbf{C}_{addr}]_p[\mathsf{gas}]_b \equiv \sigma^*[I_a]_p[\mathsf{gas}]_b - \mathsf{T}_g \times \mathsf{T}_p \tag{96}$$

- Otherwise, the sender $\mathsf{S}(\mathsf{T})$ is required to pay for the gas consumption. The balance of $\mathsf{S}(\mathsf{T})$ should satisfy $\sigma^*[\mathsf{S}(\mathsf{T})]_b \geq \mathsf{T}_g \times \mathsf{T}_p + \mathsf{T}_v$ and otherwise a *not enough balance exception* is generated. The handling of *not enough balance exception* will be discussed later. The world-state after the gas consumption payment is defined as:

$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{97}$$

$$\sigma^{**}[\mathsf{S}(\mathsf{T})]_b \equiv \max\left\{\sigma^*[\mathsf{S}(\mathsf{T})]_b - \mathsf{T}_g \times \mathsf{T}_p, 0\right\} \tag{98}$$

Storage limit validation. After charging, Conflux decides who is responsible for storage collateral. If T is eligible for sponsorship on storage collateral and calling contract $\mathbf{C} = \mathsf{T}_a$ has enough **sponsor balance for collateral**, contract $\mathbf{C}$ is responsible for the storage collateral resulted in the execution of T and will be the owner of modified entries. Otherwise, the sender $\mathsf{S}(\mathsf{T})$ is the owner of modified entries and has the obligation to pay corresponding storage collateral.

$$\mathsf{ColOwner}(\sigma,\mathsf{T}) \equiv \begin{cases} \mathsf{T}_a & \text{if } \mathsf{ColSpr}(\sigma,\mathsf{T}) = \mathsf{True} \\ \mathsf{S}(\mathsf{T}) & \text{if } \mathsf{ColSpr}(\sigma,\mathsf{T}) = \mathsf{False} \end{cases} \tag{99}$$

If $\mathsf{S}(\mathsf{T})$ is the storage owner but his balance cannot afford the full collateral as declared in **storageLimit** after transferring value T_v, i.e. $\sigma^{**}[\mathsf{S}(\mathsf{T})]_b < \mathsf{T}_v + \mathsf{T}_\ell \times 10^{18}/1024$, then the execution of T fails due to *not enough balance exception*.

Handling not enough balance exception. Whenever the preprocessing of T generates a *not enough balance exception* during preprocessing, the execution of T fails and there will be no further execution of T. To figure out whether this exception caused by the insufficient sponsorship balance in contract, the sender balance before transaction execution (i.e. $\sigma[\mathsf{S}(\mathsf{T})]_b$) is compared with a *minimum required balance* defined as

$$\mathsf{T}_v + (1 - \mathbb{I}(\mathsf{GasElig}(\sigma,\mathsf{T}))) \times \mathsf{T}_g \times \mathsf{T}_p + (1 - \mathbb{I}(\mathsf{ColElig}(\sigma,\mathsf{T}))) \times \mathsf{T}_\ell \times \frac{10^{18}}{1024}. \tag{100}$$

If $\sigma[\mathsf{S}(\mathsf{T})]_b$ has enough balance for *minimum required balance*, the sender $\mathsf{S}(\mathsf{T})$ is considered not responsible for the generated *not enough balance exception*. In this case, the resultant world-state σ' is reverted to σ, the nonce of sender is reset so that T is reusable. The receipt is composed as follows (where R' refers the receipt of last transaction):

$$R_u = R'_u \qquad R_f = 0 \qquad R_g = 0 \qquad R_{\mathbf{l}} = \varepsilon \tag{101}$$
$$R_z = 2 \qquad R_s = 0 \qquad R_{\mathbf{o}} = \varepsilon \qquad R_{\mathbf{i}} = \varepsilon \tag{102}$$

In other cases, sender $\mathsf{S}(\mathsf{T})$ is responsible for the exception. The resultant world-state is σ' is reverted to σ^{**} and the receipt is composed as follows if sender is non-existent. (i.e. $\sigma[\mathsf{S}(\mathsf{T})] \neq \varnothing$).

$$R_u = R'_u + \mathsf{T}_g \qquad R_f = \min\{\mathsf{T}_g \times \mathsf{T}_p, \sigma[\mathsf{S}(\mathsf{T})]_b\} \qquad R_g = \mathsf{GasElig}(\sigma,\mathsf{T}) \qquad R_{\mathbf{l}} = \varepsilon \tag{103}$$
$$R_z = 1 \qquad R_s = 0 \qquad R_{\mathbf{o}} = \varepsilon \qquad R_{\mathbf{i}} = \varepsilon \tag{104}$$

If sender $\mathsf{S}(\mathsf{T})$ is responsible for the exception and the sender is empty, the resultant world-state is σ' is reverted to σ. The receipt is composed as follows

$$R_u = R'_u \qquad R_f = 0 \qquad R_g = 0 \qquad R_{\mathbf{l}} = \varepsilon \tag{105}$$
$$R_z = 2 \qquad R_s = 0 \qquad R_{\mathbf{o}} = \varepsilon \qquad R_{\mathbf{i}} = \varepsilon \tag{106}$$

6.2.3 Execution Substate

The *transaction substate A* is a three tuple which accrues intermediate information during execution.

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{c}}) \tag{107}$$

The components of A are defined as follows:

- $A_{\mathbf{s}}$ is the self-destruct set of accounts that will be discarded upon the transaction's completion.
- $A_{\mathbf{l}}$ is the log series consisting of indexable "checkpoints" in the VM code execution, allowing light clients to track the execution of a contract.
- $A_{\mathbf{c}}$ is the set of key-value pairs for the storage collateral changes for each address. Similar with the world state, we write $A_{\mathbf{c}}[k] = \varnothing$ for the case that the key k does not exist and regard $A_{\mathbf{c}}[k] = \varnothing$ as $A_{\mathbf{c}}[k] = 0$.

The empty substate A^0, which is also the initial substate, has no self-destructs, no logs, no touched accounts, and zero refund. Formally, A^0 is defined as

$$A^0 \equiv (\varnothing, \varepsilon, \varnothing) \tag{108}$$

For any two substate A^1 and A^2, the accrued substate $A \equiv A^1 \uplus A^2$ is defined by

$$A_{\mathbf{s}} \equiv A^1_{\mathbf{s}} \cup A^2_{\mathbf{s}} \tag{109}$$
$$A_{\mathbf{l}} \equiv A^1_{\mathbf{l}} \cdot A^2_{\mathbf{l}} \tag{110}$$
$$\forall a \in \mathbb{B}_{160},\ A_{\mathbf{c}}[a] \equiv A^1_{\mathbf{c}}[a] + A^2_{\mathbf{c}}[a] \tag{111}$$

6.2.4 Type dependent execution

If transaction passes the preprocessing, then Conflux evaluates the *post-execution provisional state* σ^P from *pre-execution provisional state* σ^0 depending on the transaction type as specified in T_a: either contract creation or message call. The gas available for the proceeding computation is $g \equiv \mathsf{T}_g - g_0$, where g_0 is the intrinsic cost of T as in (53).

We define the tuple of post-execution provisional state σ^P, remaining gas g, accrued substate A and status code z:

$$(\sigma^P, g', A, z) \equiv \begin{cases} \Lambda(\sigma^0, \mathsf{S}(\mathsf{T}), \mathsf{S}(\mathsf{T}), \varepsilon, \mathsf{ColOwner}(\sigma, \mathsf{T}), g, \mathsf{T}_p, \mathsf{T}_v, \mathsf{T}_{\mathbf{i}}, 0, \zeta, \top) & \mathsf{T}_a = \varnothing \\ \Theta(\sigma^0, \mathsf{S}(\mathsf{T}), \mathsf{S}(\mathsf{T}), \mathsf{T}_a, \varepsilon, \mathsf{ColOwner}(\sigma, \mathsf{T}), \mathsf{T}_a, g, \mathsf{T}_p, \mathsf{T}_v, \mathsf{T}_v, \mathsf{T}_{\mathbf{d}}, 0, \top) & \mathsf{T}_a \neq \varnothing \end{cases} \tag{112}$$

Notice that we have three more parameters compared with Ethereum.

The specifications of function Λ and Θ are given in Section 6.3 and Section 6.4 respectively.

6.2.5 Postprocessing

Storage collateral refund and charge. After the message call or contract creation is processed, Conflux checks whether the incremental storage exceeds storage limit specified in T_ℓ and if the storage owner has enough balance for storage collateral. Let $i \equiv \mathsf{ColOwner}(\sigma, \mathsf{T})$ be the address who owns modified storage entries and v be the available balance to pay for storage collateral, which is defined as

$$v \equiv \begin{cases} \sigma^P[\mathsf{S}(\mathsf{T})]_b & \text{if } \mathsf{ColSpr}(\sigma, \mathsf{T}) = \mathsf{False} \\ \sigma^P[\mathsf{T}_a]_p[\mathsf{col}]_b & \text{if } \mathsf{ColSpr}(\sigma, \mathsf{T}) = \mathsf{True} \end{cases} \tag{113}$$

Notice that $A_{\mathbf{c}}[i]$ is the incremental storage collateral during execution. If $A_{\mathbf{c}}[i] > \min\{v, \mathsf{T}_\ell \times 10^{18}/1024\}$, then the execution fails because of not enough balance for collateral or exceeding the storage limit, and all the modified state will be reverted to σ^0, i.e. $\sigma' \equiv \sigma^0$. Let R' denote the receipt of last transaction. Then the receipt of current transaction T will be

$$R_u = R'_u + \mathsf{T}_g \qquad R_f = \mathsf{T}_g \times \mathsf{T}_p \qquad R_g = \mathsf{GasSpr}(\sigma, \mathsf{T}) \qquad R_{\mathbf{l}} = \varepsilon \tag{114}$$

$$R_z = 1 \qquad R_s = \mathsf{ColSpr}(\sigma, \mathsf{T}) \qquad R_{\mathbf{o}} = \varepsilon \qquad R_{\mathbf{i}} = \varepsilon \tag{115}$$

Otherwise Conflux charges and refunds storage collateral and transforms world-state σ^P into σ^*. We skim the self-destructed contracts here because their storage collateral have been refunded during self-destruction. The storage collateral in account state is also updated at this time.

$$\sigma^1 \equiv \sigma^P \qquad \text{except:} \tag{116}$$

$$\forall a \in \mathbb{B}_{160} \text{ with } A_{\mathbf{c}}[a] \neq 0, \tag{117}$$

$$\begin{cases} \sigma^1[a]_p[\mathsf{col}]_b \equiv \sigma^P[a]_p[\mathsf{col}]_b + f(a) & \text{if } a \text{ refers to a contract account, i.e. } \mathsf{Type}_a = [1000]_2 \\ \sigma^1[a]_b \equiv \sigma^P[a]_b + f(a) & \text{if } a \text{ refers to a normal account, i.e. } \mathsf{Type}_a = [0001]_2 \end{cases} \tag{118}$$

$$\sigma^1[a]_o \equiv \sigma^P[a]_o - f(a) \tag{119}$$

$$\sigma^1[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] \equiv \sigma^P[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] + \sum_{a \in \mathbb{B}_{160}} (f(a) + A_{\mathbf{c}}[a]) \tag{120}$$

$$\sigma^1[a_{\mathsf{stake}}]_{\mathbf{s}}[k_4] \equiv \sigma^P[a_{\mathsf{stake}}]_{\mathbf{s}}[k_4] + \sum_{a \in \mathbb{B}_{160}} A_{\mathbf{c}}[a] \tag{121}$$

where: (122)

$$a_{\mathsf{stake}} \equiv \mathsf{0x0888000000000000000000000000000000000002} \tag{123}$$

$$k_3 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{124}$$

$$k_4 \equiv [\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{125}$$

$$f(a) \equiv \min\{-A_{\mathbf{c}}[a], \sigma^P_o[a]\} \tag{126}$$

Gas fee refund. The *refundable amount of gas* $g^\dagger$ is the minimum of the *legitimately remaining gas* g' (as calculated in (112)) and a quarter of the **gasLimit** of T, i.e. $g^\dagger \equiv \min\{g', \mathsf{T}_g/4\}$. The refund of gas fee is applied on world-state σ^* and results in $\sigma' \equiv \Upsilon(\sigma, \mathsf{T})$.

$$\sigma^2 \equiv \sigma^1 \qquad \text{except:} \tag{127}$$

$$\begin{cases} \sigma^2[\mathsf{T}_a]_p[\mathsf{gas}]_b \equiv \sigma^1[\mathsf{T}_a]_p[\mathsf{gas}]_b + g^\dagger \times \mathsf{T}_p & \text{if } \mathsf{GasSpr}(\sigma,\mathsf{T}) = \mathsf{True} \\ \sigma^2[\mathsf{S}(\mathsf{T})]_b \equiv \sigma^1[\mathsf{S}(\mathsf{T})]_b + g^\dagger \times \mathsf{T}_p & \text{if } \mathsf{GasSpr}(\sigma,\mathsf{T}) = \mathsf{False} \end{cases} \tag{128}$$

Killed contract processing The sponsor mechanism and collateral mechanism bring additional tasks in contract destruction. First, for all the killed contract, we refund the collateral for code and storage entries to the corresponding owner.

$$\sigma^3 \equiv \sigma^2 \qquad \text{except:} \tag{129}$$

$$\forall a \in A_{\mathbf{s}},\ \sigma^2[a]_{\mathbf{s}} \equiv \varnothing \tag{130}$$

$$\forall a \in \mathbb{B}_{160} \text{ with } A^*_{\mathbf{c}}[a] \neq 0, \tag{131}$$

$$\begin{cases} \sigma^3[a]_p[\mathsf{col}]_b \equiv \sigma^2[a]_p[\mathsf{col}]_b + f(a) & \text{if } a \text{ refers to a contract account, i.e. } \mathsf{Type}_a = [1000]_2 \\ \sigma^3[a]_b \equiv \sigma^2[a]_b + f(a) & \text{if } a \text{ refers to a normal account, i.e. } \mathsf{Type}_a = [0001]_2 \end{cases} \tag{132}$$

$$\sigma^3[a]_o \equiv \sigma^2[a]_o - f(a) \tag{133}$$

$$\sigma^3[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] \equiv \sigma^2[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] + \sum_{a \in \mathbb{B}_{160}} (f(a) + A^*_{\mathbf{c}}[a]) \tag{134}$$

$$\sigma^3[a_{\mathsf{stake}}]_{\mathbf{s}}[k_4] \equiv \sigma^2[a_{\mathsf{stake}}]_{\mathbf{s}}[k_4] + \sum_{a \in \mathbb{B}_{160}} A^*_{\mathbf{c}}[a] \tag{135}$$

$$A' \equiv A \Cup A^* \tag{136}$$

where: (137)

$$a_{\mathsf{stake}} \equiv \mathsf{0x0888000000000000000000000000000000000002} \tag{138}$$

$$k_3 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{139}$$

$$k_4 \equiv [\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{140}$$

$$f(a) \equiv \min\{-A^*_{\mathbf{c}}[a], \sigma^2_o[a]\} \tag{141}$$

$$A^* \equiv A^0 \qquad \text{except:} \tag{142}$$

$$\forall a \in \mathbb{B}^{160}, A^*[a]_{\mathbf{c}} = -\frac{10^{18}}{1024} \times \sum_{a' \in A_{\mathbf{s}}} \left(\mathbb{I}(\sigma^2[a']_w = a) \times |\sigma^2[a']_{\mathbf{p}}| + \sum_{k \in \mathbb{B}_*} \mathbb{I}(\sigma^2[a']_{\mathbf{s}}[k]_o = a) \times 64 \right) \tag{143}$$

Then we refund the sponsor balance for all the killed contract.

$$\sigma^4 \equiv \sigma^3 \qquad \text{except:} \tag{144}$$

$$\forall a \in \mathbb{B}_{160},\ \sigma^4[a]_b \equiv \sigma^3[a]_b + \sum_{a' \in A_{\mathbf{s}}} \left(\mathbb{I}(\sigma^3[a']_p[\mathsf{col}]_a = a) \times \sigma^3[a']_p[\mathsf{col}]_b + \mathbb{I}(\sigma^3[a']_p[\mathsf{gas}]_a = a) \times \sigma^3[a']_p[\mathsf{gas}]_b \right) \tag{145}$$

$$\forall a' \in A_{\mathbf{s}},\ \sigma^4[a']_p[\mathsf{col}]_b \equiv 0 \tag{146}$$

$$\forall a' \in A_{\mathbf{s}},\ \sigma^4[a']_p[\mathsf{gas}]_b \equiv 0 \tag{147}$$

Then we delete all the contracts and burn all the rest balance

$$\sigma' \equiv \sigma^4 \qquad \text{except:} \tag{148}$$

$$\forall a \in A_{\mathbf{s}}, \sigma'[a] = \varnothing \tag{149}$$

$$\sigma'[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] \equiv \sigma^4[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] - \sum_{a \in A_{\mathbf{s}}} (\sigma[a]_b + \sigma[a]_t) \tag{150}$$

where: (151)

$$a_{\mathsf{stake}} \equiv \mathsf{0x0888000000000000000000000000000000000002} \tag{152}$$

$$k_3 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{153}$$

Transaction Receipt. Now the transaction execution is accomplished. The returning status code z denotes whether the execution succeeds or not. Supposing that R' is the receipt of last transaction, the receipt of current transaction will be as

follows:

$$\begin{aligned}
&R_u = R'_u + g' \quad R_f = (\mathsf{T}_g - g^\dagger) \times \mathsf{T}_p \quad R_g = \mathsf{GasSpr}(\sigma, \mathsf{T}) \quad R_{\mathbf{l}} = A'_{\mathbf{l}} \\
&R_z = z \qquad\quad R_s = \begin{cases} \mathsf{ColSpr}(\sigma, \mathsf{T}) & \text{if } z = 0 \\ 0 & \text{if } z = 1 \end{cases} \\
&R_{\mathbf{o}} = \mathsf{ToList}\left(\{(a, A'_{\mathbf{c}}[a]) | a \in \mathbb{B}_{160} \wedge A_{\mathbf{c}}[a] > 0\}\right) \\
&R_{\mathbf{i}} = \mathsf{ToList}\left(\{(a, -A'_{\mathbf{c}}[a]) | a \in \mathbb{B}_{160} \wedge A_{\mathbf{c}}[a] < 0\}\right)
\end{aligned} \tag{154}$$

6.3 Contract Creation

A number of intrinsic parameters are used when creating a smart contract account:

- world-state σ;
- sender s;
- original sender o;
- other recipients in call stack $\mathbf{t}$;
- storage owner i;
- available gas g;
- gas price p;
- endowment v;
- initialization code $\mathbf{i}$ as an arbitrary length byte array;
- the present depth of message-call/contraction-creation stack e;
- the salt for new account's address ζ,
  where $\zeta = \varnothing$ if the creation was caused by CREATE, and $\zeta \in \mathbb{B}_{256}$ if the creation was caused by CREATE2;
- and finally the permission to change the state w.

We define the contract creation function by Λ, which evaluates from the above parameters and modifies the state σ to a new state σ', together with the leftover gas g', the accrued substate A, the result of creation, and the output $\mathbf{o}$.

$$\left(\sigma', g', A, z, \mathbf{o}\right) \equiv \Lambda(\sigma, s, o, \mathbf{t}, i, g, p, v, \mathbf{i}, e, \zeta, w) \tag{155}$$

The address a of the account α newly created by CREATE is defined as the 4-bit contract type indicator concatenating the rightmost 156 bits (i.e. the 100-th to 255-th bit) of the Keccak hash of a zero byte, the sender address s, the little-endian 32-byte array of its account nonce and the Keccak hash of EVM code. For CREATE2 the rule is slightly different by substituting account nonce with the salt ζ and changing the leading byte before taking Keccak (following EIP-1014). Combining these two cases, the resultant address for the new contract account α is defined as follows:

$$a = A(s, \sigma[s]_n - 1, \zeta, \mathbf{i}) \equiv \begin{cases} [1000]_2 \circ \mathsf{KEC}\big([00]_{16} \circ s \circ \mathrm{LE}_{32}(\sigma[s]_n - 1) \circ \mathsf{KEC}(\mathbf{i})\big)[100 \ldots 255] & \text{if } \zeta = \varnothing \\ [1000]_2 \circ \mathsf{KEC}\big([ff]_{16} \circ s \circ \zeta \circ \mathsf{KEC}(\mathbf{i})\big)[100 \ldots 255] & \text{otherwise} \end{cases} \tag{156}$$

where $\mathrm{LE}_{32}(\cdot)$ denotes the function that expands an integer value in $[0, 2^{256} - 1]$ to a little-endian 32-byte array. Note that we use $\sigma[s]_n - 1$ since it is indeed the sender's nonce at the generation of the respective transaction or VM operation.

If $\sigma[a]_c \neq \mathsf{KEC}(\varepsilon)$, a *Contract Address Conflict* exception is triggered. Function Λ returns $(\varnothing, g, A^0, 1)$ immediately.

Otherwise, the account's nonce is initialized to one, the balance as the value passed by the contract creation transaction, the storage and code as for the empty string. The sender's balance is reduced by the transferred value (there must be enough balance or the transaction will not be executed). Thus the mutated state becomes σ^*:

$$\sigma^* \equiv \sigma \qquad \text{except:} \tag{157}$$

$$\sigma^*[a] \equiv \alpha^0 \quad \text{except: } \sigma^*[a]_n = 1 \wedge \sigma^*[a]_b = v + \sigma[a]_b \wedge \sigma^*[a]_a = o \tag{158}$$

$$\sigma^*[s] \equiv \begin{cases} \varnothing & \text{if } \sigma[s] = \varnothing \wedge v = 0 \\ \sigma[s] \quad \text{except}: \sigma^*[s]_b = \sigma[s]_b - v & \text{otherwise} \end{cases} \tag{159}$$

where α^0 is the default account specified in eq. (8).

The unmentioned components of an account are initialized by default.

Finally the account α is initialized by EVM code $\mathbf{i}$ according to the execution model. Code execution may 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 messages calls can be made. As such, the code execution function Ξ evaluates to a tuple of resultant state σ^{**}, available gas remaining g^{**}, the accrued substate A and the body code $\mathbf{o}$.

$$(\sigma^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\sigma^*, g, I) \tag{160}$$

where I consists of the parameters of the execution environment as follows:

$$I_a \equiv a \tag{161}$$
$$I_o \equiv o \tag{162}$$
$$I_i \equiv i \tag{163}$$
$$I_p \equiv p \tag{164}$$
$$I_{\mathbf{d}} \equiv \varepsilon \tag{165}$$
$$I_{\mathbf{t}} \equiv \mathbf{t} \tag{166}$$
$$I_s \equiv s \tag{167}$$
$$I_v \equiv v \tag{168}$$
$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{169}$$
$$I_{\mathsf{H}} \equiv \mathsf{H} \tag{170}$$
$$I_{\mathbf{L}} \equiv \mathbf{L} \tag{171}$$
$$I_e \equiv e \tag{172}$$
$$I_w \equiv w \tag{173}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_{H} is the block header of the present block. $I_{\mathbf{L}}$ is the list of block headers ordered in front of the current block.

Code execution depletes gas, and gas may not go below zero, thus the actual execution may exit before the code has come to a natural halting state. In this (and several other) exceptional cases (i.e. $\sigma^{**} = \varnothing \wedge \mathbf{o} = \varnothing$), we say an out-of-gas (OOG) exception has occurred: the evaluated state is set to the empty set $\varnothing$, and the entire contract creation should have no effect on the state, effectively leaving it as it was immediately prior to the attempt of the failed creation. Function Λ returns $(\varnothing, g^{**}, A^0, 1)$ immediately.

If the initialization code completes successfully, a final storage cost is charged for depositing the code. The storage cost s is proportional to the code size of the created contract and it consists of two parts:

- the code-deposit cost d charged as gas consumption:

$$d \equiv |\mathbf{o}| \times G_{codedeposit} \tag{174}$$

- a substate A^* will be generated to record the storage occupied by code size. the code size collateral will be charged in transaction post processing and will be locked during the lifetime of the created contract. (Conflux will record the owner of code in world-state and refund the collateral when the contract is destroyed):

$$A^* \equiv A^0 \quad \text{except: } A^*_{\mathbf{p}}[i] = |\mathbf{o}| \tag{175}$$

If the remaining gas cannot afford the code-deposit cost (i.e. $g^{**} < d$) or the code size exceeds 49152 bytes (i.e. $|\mathbf{o}| < 49152$), then we also declare that an exception occurs and handle it as a failed contract creation attempt. Function Λ returns $(\varnothing, g^{**}, A^0, 1)$ immediately.

If the contract creation fails for any reason, the value of the transaction is not transferred to the aborted contract, and collateral for storing the code is not locked either. If the contract creation succeeds, we formally specify the resultant state, gas, storage limit, substate, and status code by (σ', g', A', z) as follows:

$$g' \equiv g^{**} - d \tag{176}$$
$$\sigma' \equiv \sigma^{**} \quad \text{except:} \tag{177}$$
$$\sigma'[a]_c \equiv \mathsf{KEC}(\mathbf{o}) \tag{178}$$
$$\sigma'[a]_{code} \equiv (\mathbf{o}, i) \tag{179}$$

$$A^* \equiv A^0 \quad \text{except:} \tag{180}$$
$$A_{\mathbf{c}}[a] \equiv |\mathbf{o}| \times \frac{10^{18}}{1024} \tag{181}$$
$$A' \equiv A \Cup A^* \tag{182}$$
$$z \equiv 0 \tag{183}$$

In the determination of σ', the final body code for the newly created account is specified by the byte sequence $\mathbf{o}$ derived from the execution of the initialization code $\mathbf{i}$. The status code z is an indicator of whether the contract creation succeeds.

Therefore the result of contract creation is either a successfully created new contract with its endowment and collateral for storage, or no new contract and no transfer of value or collateral at all.

Subtleties. Note that while the initialization code is executing, the newly created address exists but with no intrinsic body code. Thus any message call received by it during this time causes no code to be executed. If the initialization execution ends with a SUICIDE 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 world-state may left with a zombie account. Only the administrator of such contract can destroy it by calling the internal contract described in section 8.2.

6.4 Message Call

The following intrinsic parameters are used when executing a message call:

- world-state σ;
- sender s;
- original sender o;
- recipient r;
- other recipients in call stack $\mathbf{t}$
- storage owner i
- the account c whose code is to be executed, usually the same as recipient;
- available gas g;
- gas price p;
- value v;
- input data $\mathbf{d}$ of the call, as an arbitrary length byte array;
- the present depth of message-call/contraction-creation stack e;
- and finally the permission to change the state w.

During the execution of message calls, the state and transaction substate may change, and finally an output data array $\mathbf{o}$ will be generated. In case of executing transactions (generated by external controllers) the output data $\mathbf{o}$ is ignored, however message calls (generated by internal execution process) can result further consequences due to the execution of VM-codes, especially when the message call is generated inside the execution of another message call (or transaction).

$$\left(\sigma', g', A, z, \mathbf{o}\right) \equiv \Theta\left(\sigma, s, o, r, \mathbf{t}, i, c, g, p, v, \tilde{v}, \mathbf{d}, e, w\right) \tag{184}$$

Note that we differentiate between the value to be transferred, v, from the value apparent in the execution context, $\tilde{v}$, for the DELEGATECALL instruction.

We let σ^* denote the first transitional world-state, which is the same as the original state except for the value transferred from sender s to recipient r (if $s \neq r$):

$$\sigma^*[r]_b \equiv \sigma[r]_b + v \qquad \wedge \qquad \sigma^*[s]_b \equiv \sigma[s]_b - v \tag{185}$$

In particular, if $\sigma[r]$ was undefined in σ, Conflux will treat it as an empty account with address r which has no code or state and zero balance and nonce. If furthermore the transferred value v is positive, the account will be created and stored in $\sigma^*[r]$. Thus the previous equation should be taken to mean:

$$\sigma^* \equiv \sigma \qquad \text{except:} \tag{186}$$

$$\sigma^*[s] \equiv \begin{cases} \varnothing & \text{if } \sigma[s] = \varnothing \wedge v = 0 \\ \sigma[s] \quad \text{except:} \sigma^*[s]_b = \sigma[s]_b - v & \text{otherwise} \end{cases} \tag{187}$$

$$\sigma^*[r] \equiv \begin{cases} \alpha^0 \quad \text{except: } \sigma^*[r]_b = v & \text{if } \sigma[r] = \varnothing \wedge v \neq 0 \\ \varnothing & \text{if } \sigma[r] = \varnothing \wedge v = 0 \\ \sigma[r] \quad \text{except:} \sigma^*[r]_b = \sigma[r]_b + v & \text{otherwise} \end{cases} \tag{188}$$

The recipient's associated code $\mathbf{b}$, whose Keccak hash is $\sigma[c]_c$, is executed according to the execution model. Note that the code $\mathbf{b}$ is stored in code component $\sigma[c]_{code}$ of account c.

Similar as with contract creation, if the execution halts due to an exception, then the state is reverted to the point immediately prior to balance transfer (i.e. σ) of the message call but no gas is refunded. The new state σ' after executing this message call is

as follows:

$$\sigma' \equiv \begin{cases} \sigma & \text{if } \sigma^{**} = \varnothing \\ \sigma^{**} & \text{otherwise} \end{cases} \tag{189}$$

$$g' \equiv \begin{cases} 0 & \text{if } \sigma^{**} = \varnothing \wedge \mathbf{o} = \varnothing \\ g^{**} & \text{otherwise} \end{cases} \tag{190}$$

$$z \equiv \begin{cases} 1 & \text{if } \sigma^{**} = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{191}$$

where the resultant state σ^{**} and available gas remaining g^{**}, together with the accrued substate A and the output data $\mathbf{o}$, are determined by the code execution function Ξ evaluated on state σ^*.

$$(\sigma^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\sigma^*, g, I) \tag{192}$$

where I contains the parameters of the execution environment as follows:

$$I_a \equiv r \tag{193}$$
$$I_o \equiv o \tag{194}$$
$$I_i \equiv i \tag{195}$$
$$I_p \equiv p \tag{196}$$
$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{197}$$
$$I_{\mathbf{t}} \equiv \mathbf{t} \tag{198}$$
$$I_s \equiv s \tag{199}$$
$$I_v \equiv \tilde{v} \tag{200}$$
$$I_{\mathbf{b}} \equiv \mathbf{b} \tag{201}$$
$$I_{\mathsf{H}} \equiv \mathsf{H} \tag{202}$$
$$I_{\mathbf{L}} \equiv \mathbf{L} \tag{203}$$
$$I_e \equiv e \tag{204}$$
$$I_w \equiv w \tag{205}$$

For the frequently used functionalities such as the elliptic curve public key recovery, the SHA2-256 hash scheme, and so on, we set up eight "precompiled computation contracts" with reserved code's address $c \in \{1, 2, \ldots, 8\}$ (with type indicator $[0000]_2$). The precompiled computation contracts have no side-effect during execution. They will not generate logs, modify accounts' storage or trigger another message call. In the present implementation of Conflux these exceptional contracts are specified as in the latest version of Ethereum [3].

Conflux also introduces internal contracts for specific usage. A high-level description for the internal contracts is given in Section 8. When the recipient's address r is one of the internal contracts, Conflux processes $\Xi_{\mathsf{internal}}(\sigma^*, g, I)$ and returns $(\sigma^{**}, g^{**}, A, \mathbf{o})$. A formal definition is given in section G.

6.5 Execution Model

The execution model specifies the system state transition on input of a sequence of bytecode instructions and a small tuple of environmental data. The state transition function is formalized as a virtual state machine, which is Turing-complete except that its running time and storage space is intrinsically bounded by the limited amount of available gas and collateral for storage. For this moment we implement the well-known Ethereum Virtual Machine (EVM), and the execution model follows [3].

6.5.1 Basics

The EVM is a stack-based architecture with 256-bit word size. The stack has a maximum size of 1024 words. The memory model is a simple word-addressed byte array. The machine also has an independent storage model which is a word-addressable word array (rather than byte array for the memory). The memory is volatile and storage is steady and maintained as part of the system state. All locations in both memory and storage are initialized as zero. The program code is stored separately in a virtual ROM that is only interactable via specific instructions.

The execution of the virtual machine may reach exceptions for various reasons, including stack underflows/overflow, invalid instruction, invalid jump destination, out-of-gas and so on. Like the out-of-gas exception, the machine halts immediately and throws an exception to the execution agent, either the transaction processor or recursively the spawning execution environment, which will catch and deal with it separately.

6.5.2 Gas Consumption

The cost of execution, aka. *gas*, is charged under following circumstances:

1. the execution of instructions, where each type of instructions is assigned an intrinsic amount of gas;
2. the generation of subordinate message call or contract creation.

6.5.3 Storage Consumption

Conflux requires a fixed amount of fund, i.e. $1/16$ CFX, locked as collateral during the whole lifetime of each 64B storage entry in the world-state. This fund is locked when the entry is created, and is unlocked and returned to the owner when that entry is cleared or overwritten by someone else eventually, as described in Section 7. The interest generated by the collateral is paid to miners as specified in Section 10.2. Thus the cost of storing an entry is proportional to the time length of storage usage.

The owner of the collateral of a storage entry, which is called "the owner of that entry" for simplicity, essentially records who has written the latest content of that entry. Normally the initial owner of an entry should be the sender of the transaction that causes the creation of this entry. However, in case a contract provides the collateral on behalf of the sender, the owner will be that contract instead (see Section 8.1 for details). When a storage entry is modified in the execution of a transaction, the ownership of this entry is changed, and the old owner's collateral for that entry is replaced by the new owner's collateral.

If a storage entry is cleared from the world-state, then the corresponding collateral is unlocked and returned to the owner of that entry. We remark that there is no refund to the actor who causes the clearance, which is distinct from the gas refunding policy in Ethereum [3]. Furthermore, to ensure that unlocked collateral for storage is always returned properly, Conflux does not allow destructing any smart contract with non-zero collateral for storage.

6.5.4 Execution Environment

Besides the global system state σ and the amount of remaining gas g, the execution agent must provide the following important information used in the execution environment, as contained in the tuple I:

- I_a, the address of the account which owns the code that is executing.
- I_o, the address of the original sender who originated this execution.
- I_i, the address of the storage owner.
- I_p, the gas price designated by the transaction that originated this execution.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; in case the execution agent is a transaction T, this would be the transaction data $\mathsf{T}_{\mathbf{d}}$.
- I_s, the address of the account that invoked the code; in case the execution agent is a transaction T, this would be the transaction sender's address $\mathsf{S}(\mathsf{T})$.
- I_v, the value, in Drip, passed to the recipient's account; in case the execution agent is a transaction T, this would be the transaction value T_v.
- $I_{\mathbf{b}}$, the byte array of the machine code to be executed.
- I_{H}, the block header of the present block.
- I_e, the depth of the current message-call or contract-creation in the stack.
- I_w, the permission to make modifications to the state.
- I_σ, the original world-state right before this execution.

The state transition is defined by the execution function Ξ, which takes as input the current world-state σ, the amount of gas g, and the input I as defined above, and outputs the resultant state σ', the remaining gas g', the accrued substate A and the resultant output $\mathbf{o}$. Formally, we define it as follows:

$$\left(\sigma', g', A, \mathbf{o}\right) \equiv \Xi\left(\sigma, g, I\right) \tag{206}$$

where we recall that the accrued state A consists of the selfdestructs set $A_{\mathbf{s}}$, the log series $A_{\mathbf{l}}$, the touched accounts $A_{\mathbf{t}}$, a series of addresses recording the owners of storage occupation $A_{\mathbf{o}}$ and a series of addresses recording the owners of storage release $A_{\mathbf{e}}$ (as described in Section 6.2.3):

$$A \equiv \left(A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{t}}, A_{\mathbf{o}}, A_{\mathbf{e}}\right) \tag{207}$$

6.5.5 Execution Overview

The Ξ function is defined mostly following the Ethereum yellowpaper [3], except for a few instructions. For self-sufficiency we explain the definition of Ξ briefly.

In most practical implementations Ξ will be modeled as an iterative progression of the pair (σ, μ) comprising the world-state and the machine state. Formally, it can be recursively defined 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.

Recall that the empty sequence, denoted by ε, is not equal to the empty set, denoted by $\varnothing$; this is important when interpreting the output of H, which evaluates to $\varnothing$ when execution is to continue but a series (potentially empty) when execution should halt.

$$\Xi(\sigma, g, I) \equiv (\sigma', \mu'_{\mathrm{g}}, A, \mathbf{o}) \tag{208}$$
$$(\sigma', \mu', A, ..., \mathbf{o}) \equiv X\big((\sigma, \mu, A^0, I)\big) \tag{209}$$
$$\mu_{\mathrm{g}} \equiv g \tag{210}$$
$$\mu_{\mathrm{pc}} \equiv 0 \tag{211}$$
$$\mu_{\mathbf{m}} \equiv (0, 0, ...) \tag{212}$$
$$\mu_{\mathrm{i}} \equiv 0 \tag{213}$$
$$\mu_{\mathbf{s}} \equiv \varepsilon \tag{214}$$
$$\mu_{\mathbf{o}} \equiv \varepsilon \tag{215}$$
$$\mu_{\mathbf{r}} \equiv \varepsilon \tag{216}$$

$$X\big((\sigma, \mu, A, I)\big) \equiv \begin{cases} \big(\varnothing, \mu, A^0, I, \varepsilon\big) & \text{if} \quad Z(\sigma, \mu, A, I) \\ \big(\varnothing, \mu', A^0, I, \mathbf{o}\big) & \text{if} \quad w = \mathsf{REVERT} \\ O(\sigma, \mu, A, I) \cdot \mathbf{o} & \text{if} \quad \mathbf{o} \neq \varnothing \\ X\big(O(\sigma, \mu, A, I)\big) & \text{otherwise} \end{cases} \tag{217}$$

where

$$\mathbf{o} \equiv H(\mu, I) \tag{218}$$
$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{219}$$
$$\mu' \equiv \mu \text{ except:} \tag{220}$$
$$\mu'_{\mathrm{g}} \equiv \mu_{\mathrm{g}} - C(\sigma, \mu, I) \tag{221}$$

Note that, when evaluating Ξ instead of X, the fourth element I' is dropped and the remaining gas μ'_{g} is extracted 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 becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until H becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

Machine State. The machine state μ is defined as the tuple $(g, \mathrm{pc}, \mathbf{m}, i, \mathbf{s}, \mathbf{r})$ which are the gas available, the program counter $pc \in \mathbb{N}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), the data stack contents and return stack contents. The memory contents $\mu_{\mathbf{m}}$ are a series of zeros of size 2^{256}. The return stack $\mu_{\mathbf{r}}$ is limited to 1023 items.

For the ease of reading, the instruction mnemonics, e.g. ADD, should be interpreted as their numeric eqdefalents; the full table of instructions and their specifics is given in Appendix E.2.

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

$$w \equiv \begin{cases} I_{\mathbf{b}}[\mu_{\mathrm{pc}}] & \text{if} \quad \mu_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \mathsf{STOP} & \text{otherwise} \end{cases} \tag{222}$$

Furthermore, we let δ and ρ denote the fixed number of stack items removed from and pushed into the data stack $\mu_{\mathbf{s}}$ by executing an instruction. Both δ and ρ are assumed subscriptable on the instruction. Similarly we define δ^* and ρ^* for the return stack $\mu_{\mathbf{r}}$, which is only accessed when entering or returning from subroutines on JUMPSUB and RETURNSUB instructions. An instruction cost function C evaluates to the full cost, in gas, of executing the given instruction.

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

$$\begin{aligned} Z(\sigma,\mu,A,I) \equiv \quad & \mu_g < C(\sigma,\mu,I) \\ \vee \quad & \delta_w = \varnothing \\ \vee \quad & \|\mu_\mathbf{s}\| < \delta_w \\ \vee \quad & (w = \mathsf{JUMP} \quad \wedge \quad \mu_\mathbf{s}[0] \notin D(I_\mathbf{b})) \\ \vee \quad & (w = \mathsf{JUMPI} \quad \wedge \quad \mu_\mathbf{s}[1] \neq 0 \quad \wedge \quad \mu_\mathbf{s}[0] \notin D(I_\mathbf{b})) \\ \vee \quad & (w = \mathsf{RETURNDATACOPY} \quad \wedge \quad \mu_\mathbf{s}[1] + \mu_\mathbf{s}[2] > \|\mu_\mathbf{o}\|) \\ \vee \quad & \|\mu_\mathbf{s}\| - \delta_w + \rho_w > 1024 \\ \vee \quad & (\neg I_\mathrm{w} \quad \wedge \quad W(w,\mu)) \end{aligned} \tag{223}$$

where

$$\begin{aligned} W(w,\mu) \equiv \quad & w \in \{\mathsf{CREATE}, \mathsf{CREATE2}, \mathsf{SSTORE}, \mathsf{SUICIDE}\} \\ \vee \quad & (\mathsf{LOG0} \leq w \wedge w \leq \mathsf{LOG4}) \\ \vee \quad & (w \in \{\mathsf{CALL}, \mathsf{CALLCODE}\} \wedge \mu_\mathbf{s}[2] \neq 0) \end{aligned} \tag{224}$$

This states that the execution is in an exceptional halting 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, if the output data size $\|\mu_\mathbf{o}\|$ is insufficient for the copy-output-data operation specified in a RETURNDATACOPY instruction, or if the new stack size would be larger than 1024 or state modification is attempted during a static call. The astute reader will realize that this implies that no instruction can, through its execution, cause an exceptional halt.

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

All such positions must be on valid instruction boundaries, 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 operations that trail it).

Formally:

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{225}$$

where:

$$D_J(\mathbf{c},i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant \|\mathbf{c}\| \\ \{i\} \cup D_J(\mathbf{c}, N(i,\mathbf{c}[i])) & \text{if} \quad \mathbf{c}[i] = \mathsf{JUMPDEST} \\ D_J(\mathbf{c}, N(i,\mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{226}$$

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

$$N(i,w) \equiv \begin{cases} i + w - \mathsf{PUSH1} + 2 & \text{if} \quad w \in [\mathsf{PUSH1}, \mathsf{PUSH32}] \\ i + 1 & \text{otherwise} \end{cases} \tag{227}$$

Normal Halting. The normal halting function H is defined:

$$H(\mu,I) \equiv \begin{cases} H_\mathrm{RETURN}(\mu) & \text{if} \quad w \in \{\mathsf{RETURN}, \mathsf{REVERT}\} \\ \varepsilon & \text{if} \quad w \in \{\mathsf{STOP}, \mathsf{SUICIDE}\} \\ \varnothing & \text{otherwise} \end{cases} \tag{228}$$

The data-returning halt operations, RETURN and REVERT, have a special function H_RETURN. Note also the difference between the empty sequence and the empty set as discussed here.

6.5.6 The Execution Cycle

Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

$$O\big((\sigma,\mu,A,I)\big) \equiv (\sigma',\mu',A',I) \tag{229}$$

$$\Delta \equiv \rho_w - \delta_w \tag{230}$$

$$\|\mu'_\mathbf{s}\| \equiv \|\mu_\mathbf{s}\| + \Delta \tag{231}$$

$$\forall x \in \big[\rho_w, \|\mu'_\mathbf{s}\| - 1\big] : \mu'_\mathbf{s}[x] \equiv \mu_\mathbf{s}[x - \Delta] \tag{232}$$

The gas is reduced by the instruction's gas cost.

$$\mu'_g \equiv \mu_g - C(\sigma, \mu, I) \tag{233}$$

For most instructions, the program counter pc increases by 1 on each cycle, except for following instructions: PUSH$*$, JUMP, JUMPI, JUMPSUB, RETURNSUB. The next valid instruction position for PUSH$*$ instructions is already specified in N as in eq. (227). We assume a function J, subscripted by one instruction from $\{\text{JUMP}, \text{JUMPI}, \text{JUMPSUB}, \text{RETURNSUB}\}$, which evaluates to the according value:

$$\mu'_{\text{pc}} \equiv \begin{cases} J_{\text{JUMP}}(\mu) & \text{if} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\mu) & \text{if} \quad w = \text{JUMPI} \\ J_{\text{JUMPSUB}}(\mu) & \text{if} \quad w = \text{JUMPSUB} \\ J_{\text{RETURNSUB}}(\mu) & \text{if} \quad w = \text{RETURNSUB} \\ N(\mu_{\text{pc}}, w) & \text{otherwise} \end{cases} \tag{234}$$

In general, we assume the memory, self-destruct set and system state do not change:

$$\mu'_{\mathbf{m}} \equiv \mu_{\mathbf{m}} \tag{235}$$
$$\mu'_{\text{i}} \equiv \mu_{\text{i}} \tag{236}$$
$$A' \equiv A \tag{237}$$
$$\sigma' \equiv \sigma \tag{238}$$

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

6.5.7 Difference from Ethereum

The execution function Ξ follows nearly the same definition as in Ethereum yellowpaper [3] except for a few instructions. When executing $O(\sigma, \mu, A, I) \equiv (\sigma', \mu', A', I')$ for the iterator function O which defines the result of a single cycle of the state machine, Conflux differs from Ethereum on following instructions.

Sub-call operations. Conflux has two additional parameters comparing with Ethereum: the recipient addresses call-state $\mathbf{t}$ and the storage owner i. In sub-call operations such as CREATE, CALL, CALLCODE, DELEGATECALL, and STATICCALL, the recipient addresses $I_{\mathbf{t}} \cdot I_a$ and storage owner I_i are passed to Λ and Θ as part of the execution environment I.

Re-entrance Protection. When calling a contract, Conflux virtual machine makes sure that re-entrance attack is impossible by preventing re-entrance message call, except the message call matches some requirements which make re-entrance attack impossible.

To be specific, the Conflux virtual machine maintains a call stack $I_{\mathbf{t}}$ and prevents the message call when the callee is already in the call stack but different from the caller before executing the code invoked by each message call. By requiring that the callee being different from the caller, it is still allowed to call and execute other functions in the caller's contract. Because in such cases the developer should be able to fully anticipate the execution flow and we do not consider it necessary to trigger the re-entrance protection.

If the message call is indeed re-entering some other contract in the call stack, the re-entrance protection is triggered unless two requirements are satisfied: a) the available gas in sub-execution of message call is no more than $G_{stipend}$, b) the message call has no call data. It means that the message call generated from solidity built-in function `send()` and `transfer()` will never trigger re-entrance protection. This design allows some widely used contract logic currently, e.g. withdrawing CFX from a contract when the transfer has to happen via a re-entrance call. When the re-entrance protection is triggered, Conflux virtual machine deals with it in the same way as "call stack overflow" or "not enough balance for value transfer". It stops calling the next contract and refunds all the available gas.

SSTORE **operation.** The SSTORE operation transforms (σ, A) into (σ', A') as follows:

$$(\sigma', A^*) \equiv \Phi(\sigma, I_a, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[1], I_i) \tag{239}$$
$$A' \equiv A \uplus A^* \tag{240}$$

where Φ is defined in section 7.1.

In Ethereum, the cost of operation SSTORE is $G_{sset} = 20000$ gas when the storage value is set to non-zero from zero, and $G_{sreset} = 5000$ gas when the storage value is set to zero. Ethereum will also refund $R_{sclear} = 15000$ gas when the storage value is set to zero from non-zero.

In Conflux, since cost of using storage is reflected by collateral for storage, there is no need to charge space consumption in gas. Thus Conflux charged $G_{sset} = 5000$ gas for all the SSTORE operation, regardless of the storage value, and there is no gas refund either. Furthermore, the Conflux ledger σ tracks the owner of every storage entry with non-zero value. The execution substate A records all changes on ownership of storage entries.

SUICIDE operation. When executing the SUICIDE operation, if the address receiving refund balance is invalid (i.e. $\mathsf{Type}_{\mu_s[0]} \bmod 2^{160} \notin \{[0000]_2, [0001]_2, [1000]_2\}$) the refund balance will be burnt. Otherwise, the SELFDESTRUCT operation transforms (σ, A) into (σ', A') following the same steps as Ethereum.

Subroutine operation. Ethereum introduces BEGINSUB, JUMPSUB, RETURNSUB instructions in EIP-2315, which is not listed in its yellowpaper. Conflux implements this instruction with identical behavior as Ethereum.

7. Collateral for Storage

Collateral for storage (CFS for short) mechanism is introduced in Conflux as a pricing method for the usage of storage, which is more fair and reasonable than the one-off storage fee in Ethereum. In principle, this mechanism requires a fund being locked as collateral for any occupation of storage space. The collateral is locked until the corresponding storage is freed or overwritten by someone else, and the corresponding interest generated by the locked collateral is assigned directly to miners for the maintenance of storage. Thus, the cost of storage in Conflux also depends on the duration of space occupation.

In Conflux, every entry of storage is 64B, which is exactly the size of a single key/value pair in the world-state. The required collateral for storage is proportional to the smallest multiple of 64B that are capable to cover all stored items. For every storage entry, the account that last writes to the entry is called *the owner of that storage entry*. If a storage entry is written in the execution of a contract **C** with sponsorship for collateral, then **C** is regarded as the account writing to that entry and hence becomes the owner accordingly (see Section 8.1 for more details). In the whole lifetime of a storage entry in the world-state, the owner of that entry must lock a fixed amount of CFX as collateral for the occupation of storage space. In particular, for each storage entry of size 64B the owner should lock $1/16$ CFX. This price is essentially 1 CFX for 1KB space, i.e. every byte of storage requires $10^{18}/1024$ Drip.

At the time that an account α becomes the owner of a storage entry (at either creation or modification), α should lock $1/16$ CFX for that entry at the end of transaction execution. If α is a normal address, the locked $1/16$ CFX is deducted from its balance. If α is a contract address, the locked $1/16$ CFX is deducted from its **sponsor balance for collateral**. If α has enough balance then the required collateral is locked automatically, otherwise if α does not have enough balance, the whole transaction execution will fail.

When a storage entry is deleted from the world-state, the corresponding $1/16$ CFX collateral is unlocked and returned to the balance of that entry's owner. In case the ownership of a storage entry is changed, the old owner's $1/16$ CFX collateral is unlocked, while the new owner must lock $1/16$ CFX as collateral at the same time.

For convenience, we introduce the function CFS which takes an account address a and a world-state σ as input and returns the total amount of Drip's of locked collateral for storage of account a in world-state σ. In case the world-state σ is clear from context, we write $\mathsf{CFS}(a)$ instead of $\mathsf{CFS}(a;\sigma)$ for succinctness.

$$\mathsf{CFS}(a) \equiv \mathsf{CFS}(a;\sigma) \equiv \sigma[a]_o \tag{241}$$

The world state also maintains the total number of locked tokens for collateral, which is stored in storage entry of staking internal contract. We introduce function ACFS to read this value from world state σ

$$\mathsf{ACFS}(\sigma) \equiv \sigma[a_{\mathsf{stake}}]_{\mathbf{s}}[k_4]_v \tag{242}$$

where: (243)

$$a_{\mathsf{stake}} \equiv 0\text{x}0888000000000000000000000000000000000002 \tag{244}$$

$$k_4 \equiv [\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{245}$$

7.1 Storage writing

In order to refund the storage collateral to payer when the storage entry is released, Conflux must track the owner for each storage entry. Here we formally describe the storage writing function $\Phi(\sigma, a, k, v, o)$, which sets storage entry k of account a to value v and address o is the storage owner. It returns updated world-state σ' and a substate A.

$$\sigma' \equiv \sigma \qquad \text{except:} \tag{246}$$

$$\sigma'[a]_{\mathbf{s}}[k] \equiv \begin{cases} (v,o) & v \neq 0 \\ \varnothing & v = 0 \end{cases} \tag{247}$$

$$A^1 \equiv A^0 \qquad \text{except:} \tag{248}$$

$$A^1[s_o]_{\mathbf{c}} \equiv -64 \times \frac{10^{18}}{1024} \quad \text{if } v \neq \sigma[a]_{\mathbf{s}}[k]_v \wedge s_o \neq s'_o \wedge s_o \neq \varnothing \tag{249}$$

$$A^2 \equiv A^0 \qquad \text{except:} \tag{250}$$

$$A^2[s'_o]_{\mathbf{c}} \equiv 64 \times \frac{10^{18}}{1024} \quad \text{if } v \neq \sigma[a]_{\mathbf{s}}[k]_v \wedge s_o \neq s'_o \wedge s'_o \neq \varnothing \tag{251}$$

$$A \equiv A^1 \uplus A^2 \tag{252}$$

$$\text{where:} \tag{253}$$

$$s \equiv \sigma[a]_{\mathbf{s}}[k] \tag{254}$$

$$s' \equiv \sigma'[a]_{\mathbf{s}}[k] \tag{255}$$

There are five special storage entries in staking vote contract 0x0888000000000000000000000000000000000002, which record the statistic information about Conflux blockchain. Their owners are always the staking vote contract and they are exempted from storage collateral. Their keys are list as follows

$$[\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{256}$$

$$[\mathsf{interest\_rate}]_{\mathsf{ch}} \tag{257}$$

$$[\mathsf{total\_staking\_tokens}]_{\mathsf{ch}} \tag{258}$$

$$[\mathsf{total\_storage\_tokens}]_{\mathsf{ch}} \tag{259}$$

$$[\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{260}$$

These five storage entries can only be accessed by the internal contract. In this document, we don't use function Φ when dealing with these entries and thus function Φ does not need to consider this special case.

8. Internal Contracts

Conflux introduces several built-in internal contracts for better system maintenance and on-chain governance. They provide solidity-like interface for developers. The interface list and their gas consumptions are list in section G.1. Section G formally describes the behavior of internal contracts. In this section, we introduce the high-level design of internal contracts.

8.1 Sponsorship for Usage of Contracts

Conflux implements a sponsorship mechanism to subsidize the usage of smart contracts. Thus, a new account with zero balance is able to call smart contracts as long as the execution is sponsored (usually by the operator of Dapps). The built-in *SponsorControl Contract* is introduced to record the sponsorship information of smart contracts.

The SponsorControl contract keeps the **SponsorInfo** information for each user-established contract **C**. The **SponsorInfo** contains the following fields.

- **sponsor for gas**: this is the account that provides the subsidy for gas consumption;
- **sponsor for collateral**: this is the account that provides the subsidy for collateral for storage;
- **sponsor balance for gas**: this is the balance of subsidy available for gas consumption;
- **sponsor balance for collateral**: this is the balance of subsidy available for collateral for storage;
- **sponsor limit for gas fee**: this is the upper bound for the gas fee subsidy paid for every sponsored transaction;

The SponsorControl contract also keeps a **whitelist** for each user-established contract **C**, which records normal accounts that are eligible for the subsidy. A special all-zero address refers to all normal accounts. If the storage entry of SponsorControl contract with key $\mathbf{C}_{addr} \cdot a$ is set to one, the address a is in the whitelist of contract **C**.

So we can check if an address a is in whitelist of contract $\mathbf{C}$ by

$$\begin{aligned}\mathsf{Whitelist}(\sigma, a, \mathbf{C}) &\equiv \sigma[a_{\mathsf{sponsor}}]_{\mathbf{s}}[\mathbf{C}_{addr} \cdot a]_v \neq 0 \ \vee \ \sigma[a_{\mathsf{sponsor}}]_{\mathbf{s}}[\mathbf{C}_{addr} \cdot y]_v \neq 0 \\ &\text{where:} \\ y &\equiv [000000000000000000000000000000000000000000]_2 \\ a_{\mathsf{sponsor}} &\equiv 0\text{x}0888000000000000000000000000000000000001 \end{aligned} \tag{261}$$

Sponsor for gas consumption. For a contract $\mathbf{C}$ with non-empty **sponsor for gas** and a transaction T calling $\mathbf{C}$, T is eligible for the subsidy for gas consumption if $\mathsf{S}(\mathsf{T})$ is in the **whitelist** of $\mathbf{C}$ and the gas fee specified by T is within the limit, i.e. $\mathsf{T}_p \times \mathsf{T}_g \leq$ **sponsor limit for gas fee** of $\mathbf{C}$. The gas consumption of T is paid from the **sponsor balance for gas** of $\mathbf{C}$ (if it is sufficient) rather than from the sender's balance, and the execution of T would fail if the **sponsor balance for gas** cannot afford the gas consumption. In case the transaction specifies a gas fee (i.e. $\mathsf{T}_p \times \mathsf{T}_g$) greater than $\min\{$**sponsor limit for gas fee**, **sponsor balance for gas**$\}$, there is no subsidy and the sender $\mathsf{S}(\mathsf{T})$ should pay for the gas consumption as usual.

Sponsor for storage collateral. For a contract $\mathbf{C}$ with non-empty **sponsor for collateral** and a transaction T calling $\mathbf{C}$, T is eligible for the subsidy for storage usage if: a) its **sender** $\mathsf{S}(\mathsf{T})$ is in the **whitelist** of $\mathbf{C}$, and b) the **sponsor balance for collateral** of $\mathbf{C}$ can afford its **storageLimit** T_ℓ (i.e. $\mathsf{T}_\ell \times 10^{18}/1024$ Drip). If T is eligible, then the collateral for storage incurred in the execution of T is deducted from **sponsor balance for collateral** of $\mathbf{C}$, and the owner of those modified storage entries is set to $\mathbf{C}$ accordingly. In case $\mathsf{S}(\mathsf{T})$ is not in the **whitelist** or the **sponsor balance for collateral** cannot cover the **storageLimit**, the sender $\mathsf{S}(\mathsf{T})$ has to pay for storage usage from its own balance as usual.

8.1.1 Sponsorship Update

Both sponsorship for gas and for collateral can be updated by calling the SponsorControl contract. The current sponsors can call this contract to transfer funds to increase the sponsor balances directly, and the current sponsor for gas is also allowed to increase the **sponsor limit for gas** by transferring ≥ 1000 times of the new limit. Other normal accounts can replace the current sponsors by calling this contract and providing more funds for sponsorship.

To replace the **sponsor for gas** of a contract $\mathbf{C}$, the new sponsor should transfer to $\mathbf{C}$ a fund more than the current **sponsor balance for gas** of $\mathbf{C}$ and set a new value for **sponsor limit for gas fee**. The new value of **sponsor limit for gas fee** should be no less than the old sponsor's limit unless the old **sponsor balance for gas** cannot afford the old limit. Moreover, the transferred fund should be ≥ 1000 times of the new limit, so that it is sufficient to subsidize at least 1000 transactions calling $\mathbf{C}$. If the above conditions are satisfied, the remaining **sponsor balance for gas** will be refunded to the old **sponsor for gas**, and then **sponsor balance for gas**, **sponsor for gas** and **sponsor limit for gas fee** will be updated according to the new sponsor's specification.

The replacement of **sponsor for collateral** is similar except that there is no analog of the limit for gas fee. The new sponsor should transfer to $\mathbf{C}$ a fund more than the fund provided by the current **sponsor for collateral** of $\mathbf{C}$. Then the current **sponsor for collateral** will be fully refunded, i.e. the sum of **sponsor balance for collateral** and $\mathsf{CFS}(\mathbf{C})$, and both collateral sponsorship fields are changed as the new sponsor's request accordingly. Note that the contract $\mathbf{C}$ is the owner of subsidized storage entries, so that the replacement of sponsorship for collateral will not affect ownership of existing storage entries.

Note: A contract account is also allowed to be a sponsor. Therefore it is possible that the sponsoring contract may be destructed before its sponsorship is replaced, in which case the receiver of sponsorship refund will be an already-destructed contract. Since the balance of such a contract is not operable (unless there is collision of KEC), it is meaningless to record that number in state and hence the refund will be burnt immediately.

8.2 Admin Management

The *AdminControl Contract* is introduced for better maintenance of other smart contracts, espeically which are generated tentatively without a proper destruction routine: it records the administrator of every user-established smart contract and handles the destruction on request of corresponding administrators.

The default administrator of a smart contract $\mathbf{C}$ is the creator of $\mathbf{C}$, i.e. the sender α of the transaction that causes the creation of $\mathbf{C}$. The current administrator of a smart contract can transfer its authority to another *normal account* by sending a request to the AdminControl contract. Contract accounts are not allowed to be the administrator of other contracts, since this mechanism is mainly for tentative maintenance. Any long term administration with customized authorization rules should be implemented inside the contract, i.e. as a specific function that handles destruction requests.

At any time, the administrator α of an existing contract $\mathbf{C}$ has the right to request destruction of $\mathbf{C}$ by calling AdminControl. If α is the current administrator of $\mathbf{C}$ and $\mathsf{CFS}(\mathbf{C}) = 0$, then the destruction request is accepted and processed as follows:

1. the balance of $\mathbf{C}$ will be refunded to α;

2. the **sponsor balance for gas** of **C** will be refunded to **sponsor for gas**;
3. the **sponsor balance for collateral** of **C** will be refunded to **sponsor for collateral**;
4. the internal state in **C** will be released and the corresponding collateral for storage refunded to owners;
5. the contract **C** is deleted from world-state.

The administrator of contract a is stored in account component a_a.

8.3 Staking Mechanism

Conflux introduces the staking mechanism for two reasons: first, staking mechanism provides a better way to charge the occupation of storage space (comparing to "pay once, occupy forever"); and second, this mechanism also helps in defining the voting power in decentralized governance.

At a high level, Conflux implements a built-in *Staking Contract* to record the staking information of all accounts. By sending a transaction to this contract, users (both external users and smart contracts) can deposit/withdraw funds, which is also called *stakes* in the contract. The interest of staked funds is issued at withdrawal, and depends on both the amount and staking period of the fund being withdrawn.

In Conflux, the staking contract keeps track of staked funds and freezing rules. For every account α the staking contract records the following:

- **staking funds**: each staking fund entry consists of the balance $v \in \mathbb{N}_{256}$ and creation time $t \in \mathbb{N}_{64}$ of a staked fund from the sender α, and the entry is cleared when the fund is completely withdrawn;
- **freezing rules**: each freezing rule entry is a combination of $(v,t) \in \mathbb{N}_{256} \times \mathbb{N}_{64}$ which promises that the total stake balance of account α must be at least v (measured in Drip) as long as the block number (as defined in BlockNo) does not exceed t. Expired freezing rule entries are cleared at the next update of freezing rules of the same account α.

Both kinds of entries in the staking contract requires collateral for storage, and the collateral is returned at the clearance of corresponding entries.

8.3.1 Interest Rate

The staking interest rate is currently set to 4% per year. Compound interest is implemented in the granularity of blocks. So the annualized interest rate is about 4.08%.

When executing a transaction sent by account α at block B to withdraw a fund of value v deposited at block B', the interest is calculated as follows:

$$\text{Interest issued to } \alpha \equiv \left\lfloor v \times \frac{f^{(\mathsf{BlockNo}(\mathsf{B}))}(n)}{f^{(\mathsf{BlockNo}(\mathsf{B}'))}(n)} \right\rfloor - v \tag{262}$$

$$\text{where:} \tag{263}$$

$$f(x) \equiv \left\lfloor x \times \left(1 + \frac{4\%}{63072000}\right) \right\rfloor \tag{264}$$

$$n \equiv 63072000 \times 2^{80} \tag{265}$$

The interest is approximately equals to

$$\left(\left(1 + \frac{4\%}{63072000}\right)^{T} - 1\right) \times v, \tag{266}$$

where $T \equiv \mathsf{BlockNo}(\mathsf{B}) - \mathsf{BlockNo}(\mathsf{B}')$ is the staking period measured by number of blocks, and 63072000 is the expected number of blocks generated in 365 days with the target block time 0.5 seconds. Therefore after the withdrawal, α's total amount of staking funds is decreased by v, and its balance is increased by:

$$\Delta(\alpha_b) \equiv v + \text{Interest issued to } \alpha \tag{267}$$

The account α only specifies the value v in its withdrawal request. The withdrawal always starts from the earliest staked fund and recursively continues to the next one until the accumulative amount is sufficient.

The same interest rate applies to collateral for storage as well, but the CFS interest is issued directly to the miners as the payment for storage usage at the generation of every new block. More details about CFS interest issuance are specified in Section 10.2.

8.3.2 Staking for Voting Power

For decentralized governance, the staking mechanism provides a way to measure the involvement and devotion of users with the new dimension of staking age.

When deciding voting power, it is fair and reasonable to take the staking time into consideration, rather than merely the amount of tokens. By relating voting power to committed staking period, the risk of being attacked is also mitigated, since the attacker must hold the tokens for a sufficiently long period to obtain enough voting power, which increases the cost of launching an attack.

For every account α, its committed staking time is recorded in the account field **staking vote info** in the form of **freezing rules**, where each entry $(v,t) \in \mathbb{N}_{256} \times \mathbb{N}_{64}$ is a promise that the staking balance of α must be at least v Drip until the index of block in total order (e.g. BlockNo) reaches t. A withdrawal request from α is invalid if any freezing rule is violated after fulfilling that request. The list of freezing rules for every account is append-only so that committed staking period cannot be canceled or shorten. However every single rule will eventually expire as the block number grows, e.g. the rule (v,t) expires at block B with $\mathsf{BlockNo}(\mathsf{B}) \geq t$. Expired freezing rule entries are cleared from state storage of the built-in staking contract at the next update of freezing rules of the same account.

Note that freezing rules are decoupled from specific staking funds, i.e. old funds can be withdrawn as long as the remaining staking balance is sufficient. Therefore, the staking contract is allowed to maintain staked funds in a first-in-first-out manner (i.e. the earliest staked fund is also first withdrawn).

The voting power of each staked token is defined in the following table:

Remaining Committed Staking Time	Voting Power
One year or more (i.e. $\geq$ 63072000 blocks)	1
Six months to one year ($\geq$ 31536000 but $<$ 63072000 blocks)	0.5
Three to six months ($\geq$ 15768000 but $<$ 31536000 blocks)	0.25
Less than three month (i.e. $<$ 15768000 blocks)	0

Therefore the total voting power of each account can be easily calculated from its freezing rules as recorded in the staking contract.

9. Proof of Work

Conflux applies the *Multi-point Ethash* function MpEthash as the Proof-of-Work function PoW. MpEthash is a twisted version of Ethash function with additional evaulation of a polynomial on multiple points. The detailed specification of this function is in Appendix F.

The MpEthash function is defined as:

$$\mathsf{MpEthash}(\mathsf{H}) \equiv \mathsf{MpEthash}\left(\mathsf{KEC}\left(\mathsf{RLP}(\mathsf{H}_{-n})\right), \mathsf{H}_n, \mathbf{d}\right) \equiv \mathsf{KEC}\left(\mathbf{s}_{\mathrm{h}} \circ \mathbf{m}_{\mathrm{c}}\right) \tag{268}$$

where $\mathbf{d}$ denotes the dataset derived from H as in Appendix F.3.3 and H_{-n} denotes the header excluding the **nonce** field, i.e. $\mathsf{H} \equiv \mathsf{H}_{-n} \circ \mathsf{H}_n$ since **nonce** is indeed the last field in the structure of block header, and the fields of H_{-n} are RLP-serialized according to their order in Section 3.4.

The output of MpEthash is the Keccak-256 hash of the concatenation of the seed hash $\mathbf{s}_{\mathrm{h}} \in \mathbb{B}_{512}$ and the compressed mix $\mathbf{m}_{\mathrm{c}} \in \mathbb{B}_{256}$. See Appendix F.4 for more details.

The MpEthash function is essentially the Proof-of-Work function PoW:

$$\mathsf{PoW}(\mathsf{H}) \equiv \mathsf{MpEthash}\left(\mathsf{H}\right) \tag{269}$$

9.1 Proof-of-Work Quality

The *proof-of-work quality* (a.k.a. *PoW quality* or simply *quality*) of a block refers to the expected amount of work spent in finding such a block. Given a block B with header $\mathsf{H}(\mathsf{B})$ and the 256-bit scalar $\mathsf{OFFSET}(\mathsf{H}) \equiv [\mathsf{H}_n(\mathsf{B})[1\ldots127]]_2 \times 2^{128} \in \mathbb{N}_{256}$ which denotes the offset of proof-of-work validation, the quality of B essentially represents the expected number of random trials to find a block B' with header H' satisfying that $\mathsf{PoW}(\mathsf{H}')$ is in between of $\mathsf{OFFSET}(\mathsf{H})$ and $\mathsf{PoW}(\mathsf{H})$.

More specifically, the block B with header $\mathsf{H}(\mathsf{B})$ has quality

$$\mathsf{QUALITY}(\mathsf{B}) \equiv \mathsf{QUALITY}(\mathsf{H}) \equiv \begin{cases} \left\lfloor 2^{256} / \left(\mathsf{PoW}(\mathsf{H}) - \mathsf{OFFSET}(\mathsf{H}) + 1\right)\right\rfloor & \text{if } \mathsf{PoW}(\mathsf{H}) > \mathsf{OFFSET}(\mathsf{H}) \\ \left\lfloor 2^{256} / \left(2^{256} + \mathsf{PoW}(\mathsf{H}) - \mathsf{OFFSET}(\mathsf{H}) + 1\right)\right\rfloor & \text{if } \mathsf{PoW}(\mathsf{H}) < \mathsf{OFFSET}(\mathsf{H}) \\ 2^{256} - 1 & \text{if } \mathsf{PoW}(\mathsf{H}) = \mathsf{OFFSET}(\mathsf{H}) \end{cases} \tag{270}$$

9.2 Difficulty Adjustment

The difficulty is adjusted according to the block generation rate in the past. More specifically, we estimate the current computing power of all miners from the number of blocks in the last 5000 epochs and the average timestamps of blocks in the beginning and ending epochs, and then set the target difficulty for the next 5000 epochs such that the expected block generation rate should be roughly one block per 0.5 seconds.

Formally, for $0 \leq j \leq 5000$, the target difficulty of a block at height j is set to $\mathbf{d}_0 \equiv 3 \times 10^4 = 30000$; for any positive integer $i \geq 1$, the target difficulty of blocks at height $j \in [5000i+1, 5000i+5000]$ is set to $\mathbf{d}_i \in \mathbb{N}_{256}$ such that

$$\mathbf{d}_i \equiv \begin{cases} \lfloor \mathbf{d}_{i-1} \times 1.5 \rfloor & \text{if } \mathbf{d}'_i > \mathbf{d}_{i-1} \times 1.5 \\ \lceil \mathbf{d}_{i-1} \times 0.5 \rceil & \text{if } 30000 \leq \mathbf{d}'_i < \mathbf{d}_{i-1} \times 0.5 \\ 30000 & \text{if } \mathbf{d}'_i < 30000 \\ \mathbf{d}'_i & \text{otherwise } (\mathbf{d}_{i-1} \times 0.5 \leq \mathbf{d}'_i \leq \mathbf{d}_{i-1} \times 1.5) \end{cases} \tag{271}$$

where $\mathbf{d}'_i \in \mathbb{N}_{256}$ is the estimation of ideal target difficulty defined as follows

$$\mathbf{d}'_i \equiv \mathbf{d}_{i-1} \times 500000 \times \left(\sum_{j=5000(i-1)+1}^{5000i} |\textsc{Epoch}_j| - 1 \right) / \left(\mathsf{H}\left(\mathsf{B}^{(5000i)}\right)_s - \min_{5000(i-1) \leq j < 5000i \wedge \mathsf{H}\left(\mathsf{B}^{(j)}\right)_s \neq 0} \left\{ \mathsf{H}\left(\mathsf{B}^{(j)}\right)_s \right\} \right) \tag{272}$$

where for every $k \in \mathbb{N}$, $\textsc{Epoch}_k$ denotes the set of *fully valid* blocks in the k-th epoch and $\mathsf{B}^{(k)}$ denotes the pivot block in $\textsc{Epoch}_k$. In the above formula the total number of blocks in last 5000 epochs is decreased by 1, which leads to an unbiased estimation of block generation rate.

Note that a single block B may not have a global view. Indeed, the best it could do is to compute the target difficulty $\mathbf{d}_i$ from its local view of blocks in $\mathsf{PAST}(\mathsf{B})$. In particular, a block B at height $h \equiv \mathsf{H}(\mathsf{B})_h$ should have target difficulty

$$\mathsf{H}(\mathsf{B})_d \equiv \begin{cases} \mathbf{d}_0 & h = 0 \\ \mathbf{d}_{\lfloor \frac{h-1}{5000} \rfloor} & h > 0, \mathbf{d}_{\lfloor \frac{h-1}{5000} \rfloor} \text{ is calculated with respect to } \mathsf{PAST}(\mathsf{B}) \end{cases} \tag{273}$$

Epoch difficulty. As soon as all nodes agree on the pivot block $\mathsf{B}^{(k)}$ at the k-th epoch $\textsc{Epoch}_k$, we can uniquely define the target difficulty of $\textsc{Epoch}_k$ as the target difficulty of $\mathsf{B}^{(k)}$. Formally,

$$\mathbf{d}_{\textsc{Epoch}_k} \equiv \mathsf{H}\left(\mathsf{B}^{(k)}\right)_d \tag{274}$$

where $\mathsf{H}\left(\mathsf{B}^{(k)}\right)_d$ is the **difficulty** field in $\mathsf{B}^{(k)}$'s header and it equals to $\mathbf{d}_{\lfloor \frac{k-1}{5000} \rfloor}$ derived from the past view of $\mathsf{B}^{(k)}$.

10. Incentive Mechanism

Conflux miners get paid by Conflux coins from two sources: the newly minted Conflux coins as block award, and the fees paid by transaction senders. In this section we specify the mechanism design for incentivizing Conflux miners. The adaptive weight introduced by the GHAST rule only affects the distribution of the first part of block award.

10.1 Base Block Award

The amount of coins issued to miners in every block is set in accordance to a global parameter which follows the mining schedule. We refer to the global parameter as the *base block award* or simply *base award*, and denote it by R_{base}.

The base block award starts at $\mathsf{R}_{base}(\mathsf{G}) = 7$ CFX per block. The based block award is adjusted in granularity of quarter. (i.e., 15768000 blocks, 91.25 days in expectation). In the first 16 quarters (i.e. roughly 4 years), the base block award is the initial value 7 CFX per block. In the next 32 quarters (i.e. roughly 8 years), it decreases by $\sqrt[32]{1/4}$ (about 4.2%) each quarter. Eventually the block reward is fixed at 1.75 CFX per block and annual inflation rate of mining issuance is below 2 percent.

For every pivot block B, the base award is defined as follows:

$$\mathsf{R}_{base}(\mathsf{B}) \equiv \begin{cases} 7\times 10^{18} & \text{if } \mathsf{QRT}(\mathsf{B}) < 16 \\ \left\lfloor 7\times 10^{6}\times (1/4)^{(\mathsf{QRT}(\mathsf{B})-16)/32} \right\rfloor \times 10^{12} & \text{if } 16 \le \mathsf{QRT}(\mathsf{B}) < 48 \\ 1.75\times 10^{18} & \text{if } \mathsf{QRT}(\mathsf{B}) \ge 48 \end{cases} \tag{275}$$

where: (276)

$$\mathsf{QRT}(\mathsf{B}) \equiv \left\lfloor \frac{|\mathsf{PAST}(\mathsf{B})|}{15768000} \right\rfloor \tag{277}$$

For every non-pivot block B, the base award $\mathsf{R}_{base}(\mathsf{B})$ of B equals to the base award of the pivot block of the epoch that B belongs to, i.e.

$$\mathsf{R}_{base}(\mathsf{B}) \equiv \mathsf{R}_{base}(\mathsf{PIVOT}(\mathsf{B}))$$

Based on $\mathsf{R}_{base}(\mathsf{B})$, Conflux defines the the actual block award issued to the author of block B with adjustments as described in the rest of Section 10.1.

10.1.1 Anti-cone Penalty

For every block B, we recall that a block B' is in the anti-cone of B if there is no directed path between B' and B, which means the chronological order of these two blocks is not reflected by the underlying Tree-Graph. For every given Tree-Graph $\mathbf{G}$, let $\mathscr{A}(\mathsf{B};\mathbf{G})$ denote the set of all anti-cone blocks of $\mathsf{B}\in\mathbf{G}$ that appear no later than 10 epochs after[5] the epoch where B resides in. When the Tree-Graph $\mathbf{G}$ is clear from context, we write $\mathscr{A}(\mathsf{B})$ instead of $\mathscr{A}(\mathsf{B};\mathbf{G})$ for short. Formally,

$$\mathscr{A}(\mathsf{B}) \equiv \mathscr{A}(\mathsf{B};\mathbf{G}) \equiv \left\{\mathsf{B}'\in\mathbf{G} \;\middle|\; \mathsf{B}\notin\mathsf{PAST}(\mathsf{B}')\wedge\mathsf{B}'\notin\mathsf{PAST}(\mathsf{B})\wedge\mathsf{H}\left(\mathsf{PIVOT}\left(\mathsf{B}'\right)\right)_h \le \mathsf{H}\left(\mathsf{PIVOT}(\mathsf{B})\right)_h + 10\right\} \tag{278}$$

In other words, let B^{10} be the pivot block at height $\mathsf{H}(\mathsf{B}^{10})_h = \mathsf{H}\left(\mathsf{PIVOT}(\mathsf{B})\right)_h + 10$, then

$$\mathscr{A}(\mathsf{B}) \equiv \mathscr{A}(\mathsf{B};\mathbf{G}) \equiv \mathsf{PAST}(\mathsf{B}^{10})\setminus(\mathsf{PAST}(\mathsf{B})\cup\mathsf{FUTURE}(\mathsf{B};\mathbf{G})\cup\{\mathsf{B}\}) \tag{279}$$

The *anti-cone penalty factor* of B is defined as

$$\mathsf{AF}(\mathsf{B}) \equiv \max\left\{0, 1-\left(\frac{\mathsf{Weight}(\mathscr{A}(\mathsf{B}))/\mathbf{d}_{\text{EPOCH}(\mathsf{B})}}{\gamma}\right)^2\right\} \tag{280}$$

where $\gamma\equiv 100$ is a fixed constant and $\mathsf{Weight}(\mathscr{A}(\mathsf{B})) \equiv \sum_{\mathsf{B}'\in\mathscr{A}(\mathsf{B})}\mathsf{Weight}(\mathsf{B}')$ refers to the total adapted weight of blocks in the anti-cone set $\mathscr{A}(\mathsf{B})$. We remark that $\mathsf{Weight}(\mathscr{A}(\mathsf{B}))/\mathbf{d}_{\text{EPOCH}(\mathsf{B})}$ is the equivalent number of blocks in the anti-cone of B, which corresponds to the portion of computing power in B's anti-cone.

This anti-cone penalty factor is introduced to incentivize inclusion of referee blocks as well as fast propagation. It also punishes withholding attacks when the blocks are not broadcast immediately. There is no additional award for referencing referee blocks, nor discount in block award for non-pivot blocks.

10.1.2 Base Factor

For convenience, we introduce the *base factor* $\mathsf{BF}(\mathsf{B})$ to indicate whether the author of B is eligible to receive any award.

If a block B in EPOCH_k has a lower target difficulty, i.e. $\mathsf{B}_d < \mathbf{d}_{\text{EPOCH}_k}$, then we decide the base award of B by its block quality $\mathsf{QUALITY}(\mathsf{B})$: it gets normal base award if the block quality $\mathsf{QUALITY}(\mathsf{B})$ reaches the epoch's target difficulty $\mathbf{d}_{\text{EPOCH}_k}$, and zero base award $\mathsf{R}_{base}(\mathsf{B})\equiv 0$ in case the quality $\mathsf{QUALITY}(\mathsf{B})$ does not meet $\mathbf{d}_{\text{EPOCH}_k}$. Note that the expected base award is effectively the same as setting $\mathsf{R}_{base}(\mathsf{B}) \equiv \frac{\mathsf{B}_d}{\mathbf{d}_{\text{EPOCH}_k}}\cdot\mathsf{R}_{base}(\text{EPOCH}_k)$.

If a block B is partially valid, blamed, or has a large anti-cone, then the author of B must have made some mistake and hence he is not eligible for any award.

Thus, the base factor $\mathsf{BF}(\mathsf{B})$ of block B is defined as

$$\mathsf{BF}(\mathsf{B}) \equiv \begin{cases} 1 & \mathsf{B} \text{ is valid and } \mathsf{AF}(\mathsf{B})>0, \text{ not blamed, and satisfies the target difficulty of } \text{EPOCH}_k \text{ (which requires} \\ & \mathsf{QUALITY}(\mathsf{B}) \ge 250\cdot\mathbf{d}_{\text{EPOCH}_k} \text{ if Adaptive}(\mathsf{B})=\mathsf{True} \text{ or } \mathsf{QUALITY}(\mathsf{B}) \ge \mathbf{d}_{\text{EPOCH}_k} \text{ if Adaptive}(\mathsf{B})=\mathsf{False}) \\ 0 & \text{otherwise } (\mathsf{B} \text{ is partially valid, } \mathsf{AF}(\mathsf{B})=0, \text{ blamed, or } \mathsf{QUALITY}(\mathsf{B}) \text{ does not satisfy the requirement}) \end{cases} \tag{281}$$

[5] If B is not on the pivot chain in $\mathbf{G}$, then $\mathscr{A}(\mathsf{B};\mathbf{G})$ also contains blocks appearing in earlier epochs but not referenced by B.

10.1.3 Actual Block Award to Miners

Taking all the discounts and adjustments into account, the block award assigned to the author of B is defined as follows:

$$\mathsf{R}_{block}(\mathsf{B}) \equiv \lfloor \mathsf{AF}(\mathsf{B}) \cdot \mathsf{BF}(\mathsf{B}) \cdot \mathsf{R}_{base}(\mathsf{B}) \rfloor \tag{282}$$

Remark: A non-pivot block B_1 may receive a higher block award than the pivot block B_2 in the same epoch, in case $\mathsf{Weight}(\mathscr{A}(\mathsf{B}_1)) < \mathsf{Weight}(\mathscr{A}(\mathsf{B}_2))$ and hence $\mathsf{AF}(\mathsf{B}_1) > \mathsf{AF}(\mathsf{B}_2)$.

10.2 Storage Maintenance Reward

Miners receive interest generated by collateral for storage, as payment to the cost of occupying storage space in the world-state. More specifically, the CFS interest generated by all blocks in each epoch is redistributed to authors of blocks in the epoch with respect to their actual mining block award. In particular, the CFS interest assigned to the author of block B is calculated as follows:

$$\mathsf{R}_{storage}(\mathsf{B}) \equiv \left\lfloor \sum_{\mathsf{B}' \in \mathsf{EPOCH}(\mathsf{B})} \left\lfloor \mathsf{ACFS}(\sigma(\mathsf{B}')) \times \frac{4\%}{63072000} \right\rfloor \times \frac{\mathsf{R}_{block}(\mathsf{B})}{\sum_{\mathsf{B}' \in \mathsf{EPOCH}(\mathsf{B})} \mathsf{R}_{block}(\mathsf{B}')} \right\rfloor \tag{283}$$

where $\sigma(\mathsf{B}')$ denotes the world-state at the beginning of the execution of transactions in block B', $\mathsf{ACFS}(\sigma(\mathsf{B}'))$ is the total CFS in B', 4% is the annual interest rate and 63072000 is the (expected) number of blocks in one year, and hence the value in parenthesis is the CFS interest generated by B'; and the distribution of CFS interest in $\mathsf{EPOCH}(\mathsf{B})$ is proportional to actual block awards $\mathsf{R}_{block}(\mathsf{B})$ as defined in Section 10.1.3.

Specially, if the total block reward for the whole epoch is zero, (i.e., $\sum_{\mathsf{B}' \in \mathsf{EPOCH}(\mathsf{B})} \mathsf{R}_{block}(\mathsf{B}') = 0$), the storage maintenance reward will not be distributed.

10.3 Transaction Fee Reward

If a transaction T is first executed successfully in the i-th epoch $\textsc{Epoch}_i$, then the transaction fee (for purchasing the consumed gas) of T is divided between all blocks that *properly include* T. Here "a block B properly includes a transaction T" means that: $\mathsf{T} \in \mathsf{B}$ and B belongs to $\textsc{Epoch}_i$ (the epoch that T is executed for the first time).

The transaction fee is distributed proportionally to the binary base factors of blocks as defined in (281). In particular, if the transaction T is exclusively packed in blocks with zero base factor, the transaction fee is burnt although the transaction will still be processed.

After execution of block B, we can get its receipt list $\mathbf{R}'(\mathsf{B})$ with the same length as transaction list B_{Ts}. So each transaction $\mathsf{T} \in \mathsf{B}_{\mathsf{Ts}}$ can be paired with its corresponding receipt R. The actually charged transaction fee is recorded in R_f. Recalling that $\mathsf{BF}(\mathsf{B})$ is a binary function respecting to the validity of B, the transaction fee assigned to B is defined as follows:

$$\mathsf{R}_{fee}(\mathsf{B}) \equiv \sum_{(\mathsf{T},R) \,:\, \mathsf{B} \text{ properly includes } \mathsf{T}} \left\lfloor \frac{R_f \cdot \mathsf{BF}(\mathsf{B})}{\sum_{\mathsf{B}' : \mathsf{B}' \text{ properly includes } \mathsf{T}} \mathsf{BF}(\mathsf{B}')} \right\rfloor \tag{284}$$

When there are multiple blocks properly include the transaction T, there may be a remainder for transaction fee of T. Formally, the remainder r equals to

$$r \equiv R_f - \sum_{\mathsf{B}' : \mathsf{B}' \text{ properly includes } \mathsf{T}} \mathsf{BF}(\mathsf{B}') \cdot \left\lfloor \frac{R_f \cdot \mathsf{BF}(\mathsf{B})}{\sum_{\mathsf{B}' : \mathsf{B}' \text{ properly includes } \mathsf{T}} \mathsf{BF}(\mathsf{B}')} \right\rfloor . \tag{285}$$

In case the remainder is non-zero, the blocks with r minimum block hash $\mathsf{KEC}(\mathsf{RLP}(\mathsf{B}_\mathsf{H}))$ will receive one more Drip. So the transaction fee can be exactly distributed to miners.

10.3.1 Why not distributing transaction fees among all blocks in that epoch?

One may suggest that the transaction fee of T should be shared by all valid blocks in that epoch, rather than among blocks that properly include T as described in (284). In what follows we show that our current implementation has several advantages:

- **For security and incentive compatibility:** the first priority of the incentive mechanism design is to guarantee that every rational participant will behave honestly, i.e. they should respect the consensus protocol and reference all the blocks they have observed. However, the mechanism of sharing transaction fee may bring incentives that prevent the author of a block referencing other blocks.

For example, if there is a transaction T with a considerable fee, miners may be willing to monopolize that fee even if suffering some punishment caused by a larger anti-cone. In particular, the author who packs T into a new block B would prefer not to share the transaction fee of T with others, especially when this fee is much higher than the anti-cone punishment caused by ignoring other blocks that are eligible to share the fee of T.

If the fee of T is to be shared with all other blocks in the same epoch, the author of B would *be hostile to all other unreferenced blocks*, and hence they get incentives to ignore those blocks by shirking all responsibility to network latency. Therefore, by manipulating transaction fees and causing such intentional ignorance, an attacker is able to effectively launch a partition attack.

If the fee of T is to be shared only among blocks including T, the author of B may still want to ignore other blocks with T. However, the difference is that the author of B would not be hostile to blocks without T, since he does not have to share the transaction fee of T with them. As a result, the author of B will be open to reference other observed blocks without T in B. Furthermore, referencing blocks with T in any later epoch is also safe, even if there are new transactions (other than T) with significant fees. Thus by manipulating transaction fees the attacker could merely cause a slightly longer latency instead of a significant partition.

In conclusion, sharing transaction fee of T only among blocks including T provides a stronger incentive of referencing other blocks honestly.

- **For efficiency:** the current design better discourages "free riders" who only mines on empty blocks and hence saves the effort of maintaining storage states and executing transactions. Since by producing empty blocks, those miners would lose the transaction fees completely, rather than receiving an average revenue of total fees in that epoch.
- **For fairness:** the current design indeed implements a tailored version of Shapley value, which is a fair distribution of total gains well-known in cooperative game theory.

10.4 Final Reward to Miners

The final mining reward of a block B will be added to the block author's account as specified in the **author** field $\mathsf{H}(\mathsf{B})_a$. Since the anti-cone of B may contain blocks up to 10 epochs after the EPOCH of B, the reward assigned to B effects at the end of 12 epochs after the EPOCH of B, where the balance of B_a is updated. For example, if B appears in EPOCH_i, then the account B_a receives the mining reward at the end of EPOCH_{i+12}. The total mining reward of B is calculated as follows:

$$\mathsf{R}(\mathsf{B}) \equiv \mathsf{R}_{block}(\mathsf{B}) + \mathsf{R}_{storage}(\mathsf{B}) + \mathsf{R}_{fee}(\mathsf{B}) \tag{286}$$

In particular, note that $\mathsf{AF}(\mathsf{B}) = 0$ implies $\mathsf{BF}(\mathsf{B}) = 0$ and $\mathsf{R}_{block}(\mathsf{B}) = \mathsf{R}_{storage}(\mathsf{B}) = \mathsf{R}_{fee}(\mathsf{B}) = 0$. Thus $\mathsf{R}(\mathsf{B}) = 0$ as long as B has a large anti-cone such that $\mathsf{AF}(\mathsf{B}) = 0$.

11. Concrete Protocol Implementation

Concretely, we set the following parameters for Conflux.

Parameter	Value
Block time	0.5 s
Maximum block size bound	200 KB
Starting coinbase award	7 CFX
Starting difficulty ($\mathbf{d}_0$)	$3 \times 10^4 = 30000$
Starting block gas limit	$3 \times 10^7 = 30000000$
Anti-cone penalty factor (γ)	100
Deferred execution	5 epochs
Mining reward freezing time	12 epochs
Snapshot period	100000 epochs
AdminControl Contract Address	0x0888000000000000000000000000000000000000
SponsorControl Contract Address	0x0888000000000000000000000000000000000001
Staking Contract Address	0x0888000000000000000000000000000000000002

In Conflux we use KEC as the collision-resistant hash function unless otherwise explicitly specified.

For authentication in the current version of Conflux, we use the same recoverable ECDSA signature scheme as in Ethereum [3]. This method utilizes the SECP-256k1 curve.

References

[1] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[2] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger (byzantium version dbc2f9b). *https://ethereum.github.io/yellowpaper/paper.pdf*, 2019-03-28.

[4] Sergio Demian Lerner. Strict Memory Hard Hashing Functions, 2014.

[5] Phong Vo Glenn Fowler, Landon Curt Noll. Fowler–Noll–Vo hash function, 1991.

[6] Jean-Philippe Aumasson and Daniel J Bernstein. Siphash: a fast short-input prf. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

Appendix A. Checklist for porting EVM contract to Conflux

The Ethereum contract is also a valid Conflux contract. So Ethereum contracts can be ported to Conflux easily and have almost same execution results. But notice that Conflux may have different behavior in the following points:

- **Gas used and refund:** Conflux requires less gas in SSTORE operation but no longer refunds resetting storage and contract destruction.
- **Gas fee refund:** Conflux will refund at most 1/4 of gas limit. So try to provide an accurate estimation for gas limit before signing transactions.
- **Contract address:** Conflux uses a different way to compute address for normal account from public key and compute contract address in contract creation. (See equation (1) and (156) for details.) The contract developers usually don't need to handle this difference.
- **Contract address conflict:** If the contract address has existed before contract creation, Conflux will abort the contract creation. This is different with the behavior in Ethereum.
- **Collateral for storage:** Conflux requires collateral for storage. Please make sure there is enough balance for storage collateral.

Appendix B. Difference between Ethereum and Conflux

	Ethereum	Conflux	
(Virtual Machine and transaction execution)			
Address type	indistinguishable for all accounts	distinct prefixes for normal (non-contract) account, *Solidity* contracts, and reserved contracts (a.k.a. "precompiled contracts")	section 3.1
Transaction field	–	added **chainID**, **storageLimit** and **epochHeight**	section 3.3
Gas consumption and refunding	all unused gas is refundable	at most a quarter of **gasLimit**	section 6.1
	full gas fee charged if execution fails on any exception	no gas fee when exception is not caused by the sender	section 6.2.2
Cost of storage	one-off gas fee	gas fee + collateral	section 6.5.3
	SSTORE costs 5000 or 20000 gas depending on the effect of executing this instruction, may cause a refund of 15000 gas for clearing a storage value	SSTORE costs G_{sset}= 5000 gas and every 64B storage costs 1/16 CFX for collateral (locked until the storage is overwritten or released)	section 7
Transaction validation	any invalid transaction leads to the whole block being invalid	invalid transactions are skipped, while other transactions in the same block can still be valid	section 4.2.5
	a transaction is invalid if sender's balance cannot afford the up-front payment for transferred value and gas fee (indeed the whole block will be invalid)	the transaction is valid if it satisfies all other assertions, but the execution fails immediately because of insufficient balance for the up-front payment (sender's nonce is increased and gas fee is charged)	section 6.2.1
	sender must pay transaction fee from his own balance (sender's balance cannot be zero)	a sponsor may pay for the cost of calling a smart contract (sender's balance can be zero)	section 8.1
	validity of a transaction cannot depend on current time or height	a transaction is only valid in a specified window of epochs	section 3.3
	no check on recipient's address	recipient address must have a valid type (i.e. normal account, Solidity contract, or reserved contract)	section 3.1
Contract creation	the address of contract created by CREATE does not depend on the initialization code	the address of contract created by both CREATE and CREATE2 depends on the initialization code	eq. (156)
	CREATE costs G_{create} = 32000, regardless initilization code length	CREATE costs the same as CREATE2	eq. (296)
	the maximum size of the byte-code is 24756 bytes	The maximum size of the byte-code is 49152 bytes	section 6.3
	on address conflict, reset contract but inherit the balance	on address conflict, abort the contract creation	section 6.3
Contract destruction	only by SUICIDE	destruction may effect on request of the contract's administrator (via the AdminControl contract)	section 8.2
Internal Contract	–	cannot be invoked by system operation, i.e. via CALLCODE or DELEGATECALL	section 8
BLOCKHASH	get the hash of one of the 256 most recent complete blocks	get the hash of the last block, return zero if querying other blocks	appendix E.2
CHAINID	EIP-1344	get the Conflux chain ID (503)	appendix E.2

Appendix C. Fee Schedule

The fee schedule G is a tuple of 35 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_{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}.
$G_{extcode}$	700	Amount of gas to pay for an EXTCODESIZE operation.
$G_{extcodehash}$	400	Amount of gas to pay for an EXTCODEHASH operation.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	5000	Paid for an SSTORE operation.
$R_{suicide}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE 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_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	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 SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ 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.
$G_{txdatanonzero}$	68	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_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for $*$COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	20	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Appendix D. Contract destruction

The contract destruction function $\Psi(\sigma, A)$ updates the world state σ and substate A and outputs (σ', A'). Formally, function Ψ is defined as follows.

$$\sigma' \equiv \sigma \qquad \text{except:} \tag{287}$$

$$\sigma'[r] \equiv \begin{cases} \varnothing & \text{if } \sigma[r] = \varnothing \ \wedge \ \sigma[I_a]_b = 0 \\ (\sigma[r]_n, \sigma[r]_b + \sigma[I_a]_b, \\ \quad \sigma[r]_{\mathbf{s}}, \sigma[r]_c) & \text{if } r \neq I_a \ \wedge \ \mathsf{Type}_r \in \{[0000]_2, [0001]_2, [1000]_2\} \\ (\sigma[r]_n, 0, \sigma[r]_{\mathbf{s}}, \sigma[r]_c) & \text{otherwise} \end{cases} \tag{288}$$

$$\sigma'[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] \equiv \sigma[a_{\mathsf{stake}}]_{\mathbf{s}}[k_3] - \sigma[I_a]_b \qquad \text{if } r = I_a \ \vee \ \mathsf{Type}_r \notin \{[0000]_2, [0001]_2, [1000]_2\} \tag{289}$$

$$\sigma'[I_a]_b \equiv 0 \tag{290}$$

where: (291)

$$r \equiv \mu_{\mathbf{s}}[0] \bmod 2^{160} \tag{292}$$

$$A' \equiv A \tag{293}$$

except: (294)

$$A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup \{I_a\} \tag{295}$$

Appendix E. Virtual Machine Specification

E.1 Gas Cost

Recalling that w denotes the current operation to be executed as in (222):

$$w \equiv \begin{cases} I_{\mathbf{b}}[\mu_{\text{pc}}] & \text{if} \quad \mu_{\text{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

The general gas cost function, C, is defined as:

$$\mathsf{C}(\sigma,\mu,I) \equiv \mathsf{C}_{mem}(\mu_{\text{i}}') - \mathsf{C}_{mem}(\mu_{\text{i}}) + \begin{cases} G_{sset} & \text{if} \quad w = \text{SSTORE} \\ G_{exp} & \text{if} \quad w = \text{EXP} \wedge \mu_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\mu_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \mu_{\mathbf{s}}[1] > 0 \\ G_{verylow} + G_{copy} \times \lceil \mu_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CALLDATACOPY} \vee \\ & \quad \text{CODECOPY} \vee \text{RETURNDATACOPY} \\ G_{extcode} + G_{copy} \times \lceil \mu_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \mu_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{log} + G_{logdata} \times \mu_{\mathbf{s}}[1] + G_{logtopic} & \text{if} \quad w = \text{LOG1} \\ G_{log} + G_{logdata} \times \mu_{\mathbf{s}}[1] + 2G_{logtopic} & \text{if} \quad w = \text{LOG2} \\ G_{log} + G_{logdata} \times \mu_{\mathbf{s}}[1] + 3G_{logtopic} & \text{if} \quad w = \text{LOG3} \\ G_{log} + G_{logdata} \times \mu_{\mathbf{s}}[1] + 4G_{logtopic} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\sigma,\mu) & \text{if} \quad w = \text{CALL} \vee \text{CALLCODE} \vee \\ & \quad \text{DELEGATECALL} \\ C_{\text{SUICIDE}}(\sigma,\mu) & \text{if} \quad w = \text{SUICIDE} \\ G_{create} + G_{sha3word} \times \lceil \mu_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CREATE} \\ G_{create} + G_{sha3word} \times \lceil \mu_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CREATE2} \\ G_{sha3} + G_{sha3word} \times \lceil \mu_{\mathbf{s}}[1] \div 32 \rceil & \text{if} \quad w = \text{SHA3} \\ G_{jumpdest} & \text{if} \quad w = \text{JUMPDEST} \\ G_{sload} & \text{if} \quad w = \text{SLOAD} \\ G_{zero} & \text{if} \quad w \in W_{zero} \\ G_{base} & \text{if} \quad w \in W_{base} \\ G_{verylow} & \text{if} \quad w \in W_{verylow} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{mid} & \text{if} \quad w \in W_{mid} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \\ G_{extcode} & \text{if} \quad w = \text{EXTCODESIZE} \\ G_{extcodehash} & \text{if} \quad w = \text{EXTCODEHASH} \\ G_{balance} & \text{if} \quad w = \text{BALANCE} \\ G_{blockhash} & \text{if} \quad w = \text{BLOCKHASH} \end{cases} \tag{296}$$

where:

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor \tag{297}$$

with C_{CALL} and C_{SUICIDE} as specified in the appropriate section below. We define the following subsets of instructions:
W_{zero} = {STOP, RETURN, REVERT}
W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, RETURNDATASIZE, POP, PC, MSIZE, GAS, CHAINID, BEGINSUB}

$W_{verylow}$ = {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, RETURNSUB}

W_{mid} = {ADDMOD, MULMOD, JUMP}

W_{high} = {JUMPI, JUMPSUB}

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.

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:

$$M(s,f,l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f+l) \div 32 \rceil) & \text{otherwise} \end{cases} \tag{298}$$

Another useful function is "all but one 64th" function L defined as:

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{299}$$

E.2 Instruction Set

As previously specified in Section 6.5, 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:

$$O(\sigma,\mu,A,I) \equiv (\sigma',\mu',A',I) \quad \text{with exceptions, as noted} \tag{300}$$

Here given are the various exceptions to the state transition rules given in Section 6.5 specified for each instruction, together with the additional instruction-specific definitions of J and C. For each instruction, also specified is ρ, the additional items placed on the data stack, and δ, the items removed from data stack, as defined in Section 6.5. For subrountine instracutions, further specified is ρ^*, the additional items pushed into the return stack, and δ^*, the items removed from return stack, as defined in Section 6.5.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted. The zero-th power of zero 0^0 is defined to be one.

Value	Mnemonic	δ	ρ	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[0] + \mu_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[0] \times \mu_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[0] - \mu_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mu_{\mathbf{s}}[1] = 0 \\ \lfloor \mu_{\mathbf{s}}[0] \div \mu_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if } \mu_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if } \mu_{\mathbf{s}}[0] = -2^{255} \wedge \mu_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\mu_{\mathbf{s}}[0] \div \mu_{\mathbf{s}}[1]) \lfloor \lvert \mu_{\mathbf{s}}[0] \div \mu_{\mathbf{s}}[1] \rvert \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mu_{\mathbf{s}}[1] = 0 \\ \mu_{\mathbf{s}}[0] \bmod \mu_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mu_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\mu_{\mathbf{s}}[0])(\lvert \mu_{\mathbf{s}}[0] \rvert \bmod \lvert \mu_{\mathbf{s}}[1] \rvert) & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mu_{\mathbf{s}}[2] = 0 \\ (\mu_{\mathbf{s}}[0] + \mu_{\mathbf{s}}[1]) \bmod \mu_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \mu_{\mathbf{s}}[2] = 0 \\ (\mu_{\mathbf{s}}[0] \times \mu_{\mathbf{s}}[1]) \bmod \mu_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x0a	EXP	2	1	Exponential operation. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[0]^{\mu_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \mu_{\mathbf{s}}[1]_{\mathrm{t}} & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\mu_{\mathbf{s}}[0] + 1) \\ \mu_{\mathbf{s}}[1]_{\mathrm{i}} & \text{otherwise} \end{cases}$

$\mu_{\mathbf{s}}[x]_{\mathrm{i}}$ gives the ith bit (counting from zero) of $\mu_{\mathbf{s}}[x]$

Value	Mnemonic	δ	ρ	Description
				10s: Comparison & Bitwise Logic Operations
0x10	LT	2	1	Less-than comparison. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] < \mu_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] > \mu_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] < \mu_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparison. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] > \mu_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparison. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] = \mu_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \mu_{\mathbf{s}}[0]_{\mathrm{i}} \wedge \mu_{\mathbf{s}}[1]_{\mathrm{i}}$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \mu_{\mathbf{s}}[0]_{\mathrm{i}} \vee \mu_{\mathbf{s}}[1]_{\mathrm{i}}$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \mu_{\mathbf{s}}[0]_{\mathrm{i}} \oplus \mu_{\mathbf{s}}[1]_{\mathrm{i}}$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} 1 & \text{if} \quad \mu_{\mathbf{s}}[0]_{\mathrm{i}} = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \mu'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \mu_{\mathbf{s}}[1]_{(i+8\mu_{\mathbf{s}}[0])} & \text{if} \quad i < 8 \wedge \mu_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ 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. $\mu'_{\mathbf{s}}[0] \equiv (\mu_{\mathbf{s}}[1] \times 2^{\mu_{\mathbf{s}}[0]}) \bmod 2^{256}$
0x1c	SHR	2	1	Logical right shift operation. $\mu'_{\mathbf{s}}[0] \equiv \lfloor \mu_{\mathbf{s}}[1] \div 2^{\mu_{\mathbf{s}}[0]} \rfloor$
0x1d	SAR	2	1	Arithmetic (signed) right shift operation. $\mu'_{\mathbf{s}}[0] \equiv \lfloor \mu_{\mathbf{s}}[1] \div 2^{\mu_{\mathbf{s}}[0]} \rfloor$ Where $\mu'_{\mathbf{s}}[0]$ and $\mu_{\mathbf{s}}[1]$ are treated as two's complement signed 256-bit integers, while $\mu_{\mathbf{s}}[0]$ is treated as unsigned.

20s: SHA3				
Value	**Mnemonic**	δ	ρ	**Description**
0x20	SHA3	2	1	Compute Keccak-256 hash. $\mu'_{\mathbf{s}}[0] \equiv \texttt{KEC}(\mu_{\mathbf{m}}[\mu_{\mathbf{s}}[0] \ldots (\mu_{\mathbf{s}}[0] + \mu_{\mathbf{s}}[1] - 1)])$ $\mu'_{\mathrm{i}} \equiv M(\mu_{\mathrm{i}}, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[1])$

				30s: Environmental Information
Value	**Mnemonic**	δ	ρ	**Description**
0x30	ADDRESS	0	1	Get address of currently executing account. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathrm{a}}$
0x31	BALANCE	1	1	Get balance of the given account. $\mu'_{\mathbf{s}}[0] \equiv \begin{cases} \sigma[\mu_{\mathbf{s}}[0]]_{\mathrm{b}} & \text{if} \quad \sigma[\mu_{\mathbf{s}}[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathrm{o}}$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathrm{s}}$ 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. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathrm{v}}$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\mu_{\mathbf{s}}[0] \ldots (\mu_{\mathbf{s}}[0]+31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \|I_{\mathbf{d}}\|$ 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. $\mu'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{d}}\|$ 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. $\forall i \in \{0 \ldots \mu_{\mathbf{s}}[2]-1\} : \mu'_{\mathbf{m}}[\mu_{\mathbf{s}}[0]+i] \equiv \begin{cases} I_{\mathbf{d}}[\mu_{\mathbf{s}}[1]+i] & \text{if} \quad \mu_{\mathbf{s}}[1]+i < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\mu_{\mathbf{s}}[1]+i$ are not subject to the 2^{256} modulo. $\mu'_{\mathrm{i}} \equiv M(\mu_{\mathrm{i}}, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[2])$ 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. $\mu'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{b}}\|$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall i \in \{0 \ldots \mu_{\mathbf{s}}[2]-1\} : \mu'_{\mathbf{m}}[\mu_{\mathbf{s}}[0]+i] \equiv \begin{cases} I_{\mathbf{b}}[\mu_{\mathbf{s}}[1]+i] & \text{if} \quad \mu_{\mathbf{s}}[1]+i < \|I_{\mathbf{b}}\| \\ \mathsf{STOP} & \text{otherwise} \end{cases}$ $\mu'_{\mathrm{i}} \equiv M(\mu_{\mathrm{i}}, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[2])$ The additions in $\mu_{\mathbf{s}}[1]+i$ are not subject to the 2^{256} modulo.
0x3a	GASPRICE	0	1	Get price of gas in current environment. $\mu'_{\mathbf{s}}[0] \equiv I_{\mathrm{p}}$ This is gas price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\mu'_{\mathbf{s}}[0] \equiv \|\mathbf{b}\|$ where $\mathtt{KEC}(\mathbf{b}) \equiv \sigma[\mu_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}}$

0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall i \in \{0 \dots \mu_\mathbf{s}[3]-1\} : \mu'_\mathbf{m}[\mu_\mathbf{s}[1]+i] \equiv \begin{cases} \mathbf{b}[\mu_\mathbf{s}[2]+i] & \text{if} \quad \mu_\mathbf{s}[2]+i < \|\mathbf{b}\| \\ \mathsf{STOP} & \text{otherwise} \end{cases}$ where $\mathtt{KEC}(\mathbf{b}) \equiv \sigma[\mu_\mathbf{s}[0] \bmod 2^{160}]_\mathrm{c}$ $\mu'_\mathrm{i} \equiv M(\mu_\mathrm{i}, \mu_\mathbf{s}[1], \mu_\mathbf{s}[3])$ The additions in $\mu_\mathbf{s}[2]+i$ are not subject to the 2^{256} modulo.
0x3d	RETURNDATASIZE	0	1	Get size of output data from the previous call from the current environment. $\mu'_\mathbf{s}[0] \equiv \|\mu_\mathbf{o}\|$
0x3e	RETURNDATACOPY	3	0	Copy output data from the previous call to memory. $\forall i \in \{0 \dots \mu_\mathbf{s}[2]-1\} : \mu'_\mathbf{m}[\mu_\mathbf{s}[0]+i] \equiv \begin{cases} \mu_\mathbf{o}[\mu_\mathbf{s}[1]+i] & \text{if} \quad \mu_\mathbf{s}[1]+i < \|\mu_\mathbf{o}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\mu_\mathbf{s}[1]+i$ are not subject to the 2^{256} modulo. $\mu'_\mathrm{i} \equiv M(\mu_\mathrm{i}, \mu_\mathbf{s}[0], \mu_\mathbf{s}[2])$
0x3f	EXTCODEHASH	1	1	Get hash of an account's code. $\mu'_\mathbf{s}[0] \equiv \begin{cases} 0 & \text{if} \quad \mathtt{DEAD}(\sigma, \mu_\mathbf{s}[0] \bmod 2^{160}) \\ \sigma[\mu_\mathbf{s}[0] \bmod 2^{160}]_\mathrm{c} & \text{otherwise} \end{cases}$

40s: Block Information

Value	Mnemonic	δ	ρ	Description
0x40	BLOCKHASH	1	1	Get the hash of the last block in block order. In Conflux, we only maintain the block hash of the previous block. When querying other block numbers, the returned result is always 0. $\mu'_\mathbf{s}[0] \equiv \begin{cases} \mathsf{KEC}(I_{\mathsf{H}_\mathbf{L}}[-1]) & \text{if} \quad \mu_\mathbf{s}[0] = \|I_{\mathsf{H}_\mathbf{L}}\| - 1 \\ 0 & \text{otherwise} \end{cases}$
0x41	COINBASE	0	1	Get the block's beneficiary address. $\mu'_\mathbf{s}[0] \equiv I_{\mathsf{H}_\mathrm{c}}$
0x42	TIMESTAMP	0	1	Get the block's timestamp. $\mu'_\mathbf{s}[0] \equiv I_{\mathsf{H}_\mathrm{s}}$
0x43	NUMBER	0	1	Get the block's index in total order. (The index of genesis block is 0.) $\mu'_\mathbf{s}[0] \equiv \|I_{\mathsf{H}_\mathbf{L}}\|$
0x44	DIFFICULTY	0	1	Get the block's difficulty. $\mu'_\mathbf{s}[0] \equiv I_{\mathsf{H}_\mathrm{d}}$
0x45	GASLIMIT	0	1	Get the block's gas limit. $\mu'_\mathbf{s}[0] \equiv I_{\mathsf{H}_\ell}$
0x46	CHAINID	0	1	Get the chain ID. $\mu'_\mathbf{s}[0] \equiv 2$
0x47	SELFBALANCE	0	1	Get balance of the currently executing account. $\mu'_\mathbf{s}[0] \equiv \begin{cases} \sigma[I_\mathrm{a}]_\mathrm{b} & \text{if} \quad \sigma[I_\mathrm{a} \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$

50s: Stack, Memory, Storage and Flow Operations				
Value	**Mnemonic**	δ	ρ	**Description**
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{m}}[\mu_{\mathbf{s}}[0] \dots (\mu_{\mathbf{s}}[0]+31)]$ $\mu'_{\mathrm{i}} \equiv \max(\mu_{\mathrm{i}}, \lceil (\mu_{\mathbf{s}}[0]+32) \div 32 \rceil)$ The addition in the calculation of μ'_{i} is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\mu'_{\mathbf{m}}[\mu_{\mathbf{s}}[0] \dots (\mu_{\mathbf{s}}[0]+31)] \equiv \mu_{\mathbf{s}}[1]$ $\mu'_{\mathrm{i}} \equiv \max(\mu_{\mathrm{i}}, \lceil (\mu_{\mathbf{s}}[0]+32) \div 32 \rceil)$ The addition in the calculation of μ'_{i} is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\mu'_{\mathbf{m}}[\mu_{\mathbf{s}}[0]] \equiv (\mu_{\mathbf{s}}[1] \bmod 256)$ $\mu'_{\mathrm{i}} \equiv \max(\mu_{\mathrm{i}}, \lceil (\mu_{\mathbf{s}}[0]+1) \div 32 \rceil)$ The addition in the calculation of μ'_{i} is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\mu'_{\mathbf{s}}[0] \equiv \sigma[I_{\mathrm{a}}]_{\mathbf{s}}[\mu_{\mathbf{s}}[0]]_v$
0x55	SSTORE	2	0	Save word and its owner to storage $(\sigma', A^*) \equiv \Phi(\sigma, I_a, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[1], I_i)$ $A' \equiv A \Cup A^*$ where Φ is defined in section 7.1
0x56	JUMP	1	0	Alter the program counter. $J_{\mathsf{JUMP}}(\mu) \equiv \mu_{\mathbf{s}}[0]$ This has the effect of writing said value to μ_{pc}. See (233) in Section 6.5.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\mathsf{JUMPI}}(\mu) \equiv \begin{cases} \mu_{\mathbf{s}}[0] & \text{if} \quad \mu_{\mathbf{s}}[1] \neq 0 \\ \mu_{\mathrm{pc}}+1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to μ_{pc}. See (233) in Section 6.5.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathrm{pc}}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\mu'_{\mathbf{s}}[0] \equiv 32\mu_i$
0x5a	GAS	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction. $\mu'_{\mathbf{s}}[0] \equiv \mu_g$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

50s: Stack, Memory, Storage and Flow Operations – Subroutine Operations

Note 1: Here we list columns of δ^* and ρ^* because JUMPSUB and RETURNSUB may change the return stack $\mu_\mathbf{r}$. However, $\mu_\mathbf{r}$ is only alterable by these two instructions, and hence there is no need to validate popped values.

Note 2: The actual state of the return stack is neither observable by EVM code nor consensus-critical to the protocol. Thus, a node implementor may code JUMPSUB to unobservably push pc on the return stack rather than $\text{pc}+1$, which is allowed so long as the next RETURNSUB would observably return control to the $\text{pc}+1$ location.

Value	Mnemonic	δ	ρ	δ^*	ρ^*	Description
0x5c	BEGINSUB	0	0	0	0	Marks the entry point to a subroutine. Attempted execution of a BEGINSUB causes an abort: terminate execution with an *Invalid Sub-entry* exception.
0x5d	RETURNSUB	0	0	1	0	Returns from a subroutine. If $\|\mu_\mathbf{r}\| = 0$, then abort: terminate execution with a *Return Stack Underflow* exception. Otherwise $J_{\text{RETURNSUB}}(\mu) \equiv \mu_\mathbf{r}[0]$ This has the effect of writing said value to μ_{pc}. See (233) in Section 6.5.
0x5e	JUMPSUB	1	0	0	1	Jumps to a defined BEGINSUB subroutine and transfers control to it. If $\|\mu_\mathbf{r}\| = 1023$, then abort: terminate execution with an *Out Of Return Stack* exception. Else if $I_\mathbf{b}[\mu_\mathbf{s}[0]] \neq$ BEGINSUB then abort: terminate execution with a *Bad Jump Destination* exception. Otherwise: $\mu'_\mathbf{r}[0] \equiv \mu_{\text{pc}} + 1$ $J_{\text{JUMPSUB}}(\mu) \equiv \mu_\mathbf{s}[0] + 1$ This has the effect of writing said value to μ_{pc}. See (233) in Section 6.5. In case $\mu_\mathbf{s}[0] + 1 \geq \|I_\mathbf{b}\|$, i.e. the resulting pc is beyond the last instruction, then the opcode is implicitly a STOP, which is not an error.

60s & 70s: Push Operations

Value	Mnemonic	δ	ρ	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\mu'_\mathbf{s}[0] \equiv c(\mu_{\text{pc}} + 1)$ where $c(x) \equiv \begin{cases} I_\mathbf{b}[x] & \text{if} \quad x < \|I_\mathbf{b}\| \\ 0 & \text{otherwise} \end{cases}$ 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. $\mu'_\mathbf{s}[0] \equiv c\big((\mu_{\text{pc}} + 1)\ldots(\mu_{\text{pc}} + 2)\big)$ with $c(x) \equiv (c(x_0), ..., c(x_{\|x\|-1}))$ 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. $\mu'_\mathbf{s}[0] \equiv c\big((\mu_{\text{pc}} + 1)\ldots(\mu_{\text{pc}} + 32)\big)$ 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	ρ	**Description**
0x80	DUP1	1	2	Duplicate 1st stack item. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplicate 2nd stack item. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplicate 16th stack item. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[15]$

90s: Exchange Operations				
Value	**Mnemonic**	δ	ρ	**Description**
0x90	SWAP1	2	2	Exchange 1st and 2nd stack items. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[1]$ $\mu'_{\mathbf{s}}[1] \equiv \mu_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Exchange 1st and 3rd stack items. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[2]$ $\mu'_{\mathbf{s}}[2] \equiv \mu_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Exchange 1st and 17th stack items. $\mu'_{\mathbf{s}}[0] \equiv \mu_{\mathbf{s}}[16]$ $\mu'_{\mathbf{s}}[16] \equiv \mu_{\mathbf{s}}[0]$

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_{\mathrm{a}}, \mathbf{t}, \mu_{\mathbf{m}}[\mu_{\mathbf{s}}[0] \ldots (\mu_{\mathbf{s}}[0] + \mu_{\mathbf{s}}[1] - 1)])$
and to update the memory consumption counter:
$\mu'_{\mathrm{i}} \equiv M(\mu_{\mathrm{i}}, \mu_{\mathbf{s}}[0], \mu_{\mathbf{s}}[1])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

Value	**Mnemonic**	δ	ρ	**Description**
0xa0	LOG0	2	0	Append log record with no topics. $\mathbf{t} \equiv \varepsilon$
0xa1	LOG1	3	0	Append log record with one topic. $\mathbf{t} \equiv (\mu_{\mathbf{s}}[2])$
⋮	⋮	⋮	⋮	⋮
0xa4	LOG4	6	0	Append log record with four topics. $\mathbf{t} \equiv (\mu_{\mathbf{s}}[2], \mu_{\mathbf{s}}[3], \mu_{\mathbf{s}}[4], \mu_{\mathbf{s}}[5])$

f0s: System operations

Value	Mnemonic	δ	ρ	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \mu_\mathbf{m}[\mu_\mathbf{s}[1] \dots (\mu_\mathbf{s}[1] + \mu_\mathbf{s}[2] - 1)]$ $\zeta \equiv \varnothing$ $(\sigma', \mu'_\mathrm{g}, A^+, \mathbf{o}) \equiv \begin{cases} \Lambda(\sigma^*, I_\mathrm{a}, I_\mathrm{o}, I_\mathbf{t} \cdot I_a, I_i, L(\mu_\mathrm{g}), I_\mathrm{p}, \mu_\mathbf{s}[0], \mathbf{i}, I_\mathrm{e}+1, \zeta, I_\mathrm{w}) & \text{if} \quad \mu_\mathbf{s}[0] \leqslant \sigma[I_\mathrm{a}]_\mathrm{b} \\ & \quad \wedge\, I_\mathrm{e} < 1024 \\ (\sigma, \mu_\mathrm{g}, \varnothing) & \text{otherwise} \end{cases}$ $\sigma^* \equiv \sigma \quad \text{except} \quad \sigma^*[I_\mathrm{a}]_\mathrm{n} = \sigma[I_\mathrm{a}]_\mathrm{n} + 1$ $\mu'_\mathbf{s}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\sigma' = \varnothing$, or $I_\mathrm{e} = 1024$ (the maximum call depth limit is reached) or $\mu_\mathbf{s}[0] > \sigma[I_\mathrm{a}]_\mathrm{b}$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_\mathrm{a}, \sigma[I_\mathrm{a}]_\mathrm{n}, \zeta, \mathbf{i})$, the address of the newly created account (156). $\mu'_\mathrm{i} \equiv M(\mu_\mathrm{i}, \mu_\mathbf{s}[1], \mu_\mathbf{s}[2])$ $\mu'_\mathbf{o} \equiv \varepsilon$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \mu_\mathbf{m}[\mu_\mathbf{s}[3] \dots (\mu_\mathbf{s}[3] + \mu_\mathbf{s}[4] - 1)]$ $(\sigma', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\sigma, I_\mathrm{a}, I_\mathrm{o}, t, I_\mathbf{t} \cdot I_a, I_i, t, C_{\text{CALLGAS}}(\mu), & \text{if} \quad p \\ \quad I_\mathrm{p}, \mu_\mathbf{s}[2], \mu_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \\ (\sigma, g, \varnothing, \varepsilon) & \text{otherwise} \end{cases}$ $p \equiv \mu_\mathbf{s}[2] \leqslant \sigma[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \wedge \mathsf{Type}_t \in \{[0000]_2, [0001]_2, [1000]_2\}$ $\wedge\, (t \notin I_\mathbf{t} \vee t = I_a \vee (C_{\text{CALLGAS}}(\sigma, \mu) \leq G_{callstipend} \wedge \mu_\mathbf{s}[4] = 0))$ $n \equiv \min(\{\mu_\mathbf{s}[6], \|\mathbf{o}\|\})$ $\mu'_\mathbf{m}[\mu_\mathbf{s}[5] \dots (\mu_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\mu'_\mathbf{o} = \mathbf{o}$ $\mu'_\mathrm{g} \equiv \mu_\mathrm{g} + g'$ $\mu'_\mathbf{s}[0] \equiv x$ $A' \equiv A \uplus A^+$ $t \equiv \mu_\mathbf{s}[1] \bmod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\sigma' = \varnothing$ or if $p = \mathsf{False}$ which means EVM prevents this call; $x = 1$ otherwise. $\mu'_\mathrm{i} \equiv M(M(\mu_\mathrm{i}, \mu_\mathbf{s}[3], \mu_\mathbf{s}[4]), \mu_\mathbf{s}[5], \mu_\mathbf{s}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_{\text{CALL}}(\sigma, \mu) \equiv C_{\text{GASCAP}}(\sigma, \mu) + C_{\text{EXTRA}}(\sigma, \mu)$ $C_{\text{CALLGAS}}(\sigma, \mu) \equiv \begin{cases} C_{\text{GASCAP}}(\sigma, \mu) + G_{callstipend} & \text{if} \quad \mu_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\sigma, \mu) & \text{otherwise} \end{cases}$ $C_{\text{GASCAP}}(\sigma, \mu) \equiv \begin{cases} \min\{L(\mu_\mathrm{g} - C_{\text{EXTRA}}(\sigma, \mu)), \mu_\mathbf{s}[0]\} & \text{if} \quad \mu_\mathrm{g} \geq C_{\text{EXTRA}}(\sigma, \mu) \\ \mu_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_{\text{EXTRA}}(\sigma, \mu) \equiv G_{call} + C_{\text{XFER}}(\mu) + C_{\text{NEW}}(\sigma, \mu)$ $C_{\text{XFER}}(\mu) \equiv \begin{cases} G_{callvalue} & \text{if} \quad \mu_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{NEW}}(\sigma, \mu) \equiv \begin{cases} G_{newaccount} & \text{if} \quad \texttt{DEAD}(\sigma, \mu_\mathbf{s}[1] \bmod 2^{160}) \wedge \mu_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$

0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\sigma', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\sigma^*, I_\mathrm{a}, I_\mathrm{o}, I_\mathrm{a}, I_\mathbf{t} \cdot I_a, I_i, & \\ \quad t, C_{\text{CALLGAS}}(\mu), I_\mathrm{p}, \mu_\mathbf{s}[2], & \text{if} \quad p \\ \quad \mu_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \\ (\sigma, g, \varnothing, \varepsilon) & \text{otherwise} \end{cases}$ where $p \equiv \mu_\mathbf{s}[2] \leqslant \sigma[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \wedge \mathsf{Type}_t \in \{[0000]_2, [0001]_2, [1000]_2\}$. Note the change in the fourth parameter to the call Θ from the 2nd stack value $\mu_\mathbf{s}[1]$ (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. $H_{\text{RETURN}}(\mu) \equiv \mu_\mathbf{m}[\mu_\mathbf{s}[0] \ldots (\mu_\mathbf{s}[0] + \mu_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 6.5. $\mu'_\mathrm{i} \equiv M(\mu_\mathrm{i}, \mu_\mathbf{s}[0], \mu_\mathbf{s}[1])$
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 $\mu_\mathbf{s}[2]$. As a result, $\mu_\mathbf{s}[3]$, $\mu_\mathbf{s}[4]$, $\mu_\mathbf{s}[5]$ and $\mu_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\mu_\mathbf{s}[2]$, $\mu_\mathbf{s}[3]$, $\mu_\mathbf{s}[4]$ and $\mu_\mathbf{s}[5]$. Otherwise it is equivalent to CALL except: $(\sigma', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\sigma^*, I_\mathrm{s}, I_\mathrm{o}, I_\mathrm{a}, I_\mathbf{t} \cdot I_a, I_i, t, C_{\text{CALLGAS}}(\mu), & \\ \quad I_\mathrm{p}, 0, I_\mathrm{v}, \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \text{if} \quad p \\ (\sigma, g, \varnothing, \varepsilon) & \text{otherwise} \end{cases}$ where $p \equiv I_\mathrm{v} \leqslant \sigma[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \wedge \mathsf{Type}_t \in \{[0000]_2, [0001]_2, [1000]_2\}$. Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. 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 $\zeta \equiv \mu_\mathbf{s}[3]$.
0xfa	STATICCALL	6	1	Static message-call into an account. Exactly equivalent to CALL except: The argument $\mu_\mathbf{s}[2]$ is replaced with 0. The deeper argument $\mu_\mathbf{s}[3]$, $\mu_\mathbf{s}[4]$, $\mu_\mathbf{s}[5]$ and $\mu_\mathbf{s}[6]$ are respectively replaced with $\mu_\mathbf{s}[2]$, $\mu_\mathbf{s}[3]$, $\mu_\mathbf{s}[4]$ and $\mu_\mathbf{s}[5]$. The last argument of Θ is $\bot$.
0xfd	REVERT	2	0	Halt execution reverting state changes but returning data and remaining gas. The effect of this operation is described in (217). For the gas calculation, we use the memory expansion function, $\mu'_\mathrm{i} \equiv M(\mu_\mathrm{i}, \mu_\mathbf{s}[0], \mu_\mathbf{s}[1])$
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SUICIDE	1	0	Halt execution and register account for later deletion. $(\sigma', A') \equiv \Psi(\sigma, A)$ where Ψ is defined in section D. $C_{\text{SUICIDE}}(\sigma, \mu) \equiv G_{suicide} + \begin{cases} G_{newaccount} & \text{if} \quad n \\ 0 & \text{otherwise} \end{cases}$ $n \equiv \mathtt{DEAD}(\sigma^*, \mu_\mathbf{s}[0] \bmod 2^{160}) \wedge \sigma[I_\mathrm{a}]_\mathrm{b} \neq 0$

Appendix F. Multi-point Evaluation Hashing

F.1 Definitions

We employ the following definitions:

Name	Value	Description
$J_{wordbytes}$	4	Bytes in word.
$J_{datasetinit}$	2^{32}	Bytes in dataset at genesis.
$J_{datasetgrowth}$	2^{24}	Dataset growth per stage.
$J_{cacheinit}$	2^{24}	Bytes in cache at genesis.
$J_{cachegrowth}$	2^{16}	Cache growth per stage.
J_{stage}	2^{19}	Epoches per stage.
$J_{cacherounds}$	3	Number of rounds in cache production.
$J_{mixbytes}$	256	mix length in bytes.
$J_{hashbytes}$	64	Hash length in bytes.
$J_{parents}$	256	Number of parents of each dataset element.
J_{pow}	1024	Number of polynomial coefficients in multiple point evaluation.
$J_{accesses}$	32	Number of accesses in hashimoto loop.
$J_{warpsize}$	32	$J_{pow}/J_{accesses}$
J_{mod}	1032193	Modulus in multiple point evaluation.

F.2 Size of Dataset and Cache

The size for the hash function's cache $\mathbf{c} \in \mathbb{B}^*$ and dataset $\mathbf{d} \in \mathbb{B}^*$ depend on the stage, which in turn depends on the block height H_h.

$$E_{stage}(\mathsf{H}_h) \equiv \left\lfloor \frac{\mathsf{H}_h}{J_{stage}} \right\rfloor \tag{301}$$

The size of the dataset growth by $J_{datasetgrowth}$ bytes, and the size of the cache by $J_{cachegrowth}$ bytes, every stage. 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 $d_{size} = \|\mathbf{d}\|$ be the size of the dataset, which is calculated using

$$d_{size} \equiv E_{\text{prime}}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{stage}, J_{mixbytes}) \tag{302}$$

The size of the cache, c_{size}, is calculated using

$$c_{size} \equiv E_{\text{prime}}(J_{cacheinit} + J_{cachegrowth} \cdot E_{stage}, J_{hashbytes}) \tag{303}$$

where $E_{\text{prime}}(x, y)$ asserts x/y is an integer and returns the largest number x' such that $x' < x$ and $x' = p \cdot y$ for some prime number p.

$$E_{\text{prime}}(x, y) \equiv \max_{x' < x \wedge \mathsf{IsPrime}(x'/y)} x' \tag{304}$$

F.3 Stage Dataset Generation

In order to generate the dataset $\mathbf{d}$ for stage J_{stage}, we need the cache $\mathbf{c}$, which is an array of bytes. It depends on the cache size c_{size} and the seed hash $\mathbf{s} \in \mathbb{B}_{256}$.

F.3.1 Seed hash

The seed hash is different for every stage. For the first stage it is the Keccak-256 hash of a series of 256 bits (32 bytes) of zeros. For every other stage it is always the Keccak-256 hash of the previous seed hash:

$$\mathbf{s} \equiv \mathsf{KEC}^{(E_{stage}(\mathsf{H}_h))}(\mathbf{0}_{256}) \tag{305}$$

where $\mathbf{0}_{256}$ denotes 256 bits of zeros. (Recalling that $f^{(n)}$ represents calling function f in n times recursively.)

F.3.2 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 [4]. The initial cache $\mathbf{c}'$ will be constructed as follows.

Recalling that KEC512 denotes the Keccak-512 hash function whose output length is 512 bits (64 single bytes), we define initial cache $\mathbf{c}'$ can be defined as:

$$\mathbf{c}'[i] \equiv \mathsf{KEC512}^{(i)}(\mathbf{s}), \quad \forall \quad i \in \{0,1,2,\ldots,n-1\} \tag{306}$$

where n denote the number of elements in cache:

$$n \equiv c_{size}/J_{hashbytes} \tag{307}$$

The cache $\mathbf{c}$, consisting of n items of KEC512 hash values, is calculated by performing $J_{cacherounds}$ rounds of the RandMemoHash algorithm to the initial cache $\mathbf{c}'$:

$$\mathbf{c} \equiv E_{\text{RMH}}^{(J_{cacherounds})}(\mathbf{c}') \tag{308}$$

Every single round of the RandMemoHash algorithm modifies each subset of the cache as follows:

$$E_{\text{RMH}}(\mathbf{x}) \equiv \big(E_{rmh}(\mathbf{x},0), E_{rmh}(\mathbf{x},1), ..., E_{rmh}(\mathbf{x},n-1)\big) \tag{309}$$

$$E_{rmh}(\mathbf{x},i) \equiv \mathsf{KEC512}(\mathbf{x}'[(i-1+n) \bmod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \bmod n]) \quad \text{with} \quad \mathbf{x}'[j] = \begin{cases} E_{rmh}(\mathbf{x},j) & j < i \\ \mathbf{x}[j] & j \geq i \end{cases} \tag{310}$$

where $\mathbf{x}'[i][0]$ denotes the first word of $\mathbf{x}'[i]$.

F.3.3 Full dataset calculation

Essentially, we combine data from $J_{parents}$ pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset $\mathbf{d}$ is then generated by a number of items, each of $J_{hashbytes}$ bytes in size:

$$\mathbf{d}[i] \equiv E_{datasetitem}(\mathbf{c},i), \quad \forall \quad 0 \leq i < d_{size}/J_{hashbytes} \tag{311}$$

In order to calculate the single item we use an algorithm $E_{\text{FNV}} : \mathbb{N}_{32} \times \mathbb{N}_{32} \to \mathbb{N}_{32}$ inspired by the FNV hash [5] in some cases as a non-associative substitute for XOR.

$$E_{\text{FNV}}(\mathbf{x},\mathbf{y}) \equiv ((\mathbf{x} \times 0\text{x}01000193) \oplus \mathbf{y}) \bmod 2^{32} \tag{312}$$

When E_{FNV} receives input in $\mathbb{B}_{32}$, it interprets it as a little-endian encoding integer in $\mathbb{N}_{32}$.

The single item of the dataset can now be calculated by iteratively mixing items from the cache $\mathbf{c}$ as follows:

$$E_{datasetitem}(\mathbf{c},i) \equiv \mathsf{KEC512}\Big(\mathbf{m}_{J_{parents}}\Big) \tag{313}$$

where $\mathbf{m}_j$ is updated from $\mathbf{m}_{j-1}$ by function E_{mix}. $\mathbf{m}_0$ is initialized with the hash value computed from cache $\mathbf{c}$ and index i.

$$\mathbf{m}_j \equiv E_{mix}(\mathbf{c},i,\mathbf{m}_{j-1},j-1), \quad \forall 1 \leq j \leq J_{parents} \tag{314}$$

$$\mathbf{m}_0 \equiv \mathsf{KEC512}(\mathbf{c}[i \bmod n] \oplus i) \tag{315}$$

$$E_{mix}(\mathbf{c},i,\mathbf{m},p) \equiv E_{\text{FNV}}^{*}\Big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p,\ \mathbf{m}\big[p \bmod J_{hashbytes}/J_{wordbytes}\big]) \bmod n]\Big) \tag{316}$$

Here, i is regarded as a 512-bit string in little-endian and E_{FNV}^{*} denotes the element-wise invocation of E_{FNV} over $J_{hashbytes}$-bit string, which is interpreted as an array of words in little-endian.

F.4 Proof-of-work function

Essentially, we maintain a "mix" of $J_{mixbytes}$ 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 MpEthash function takes $\mathsf{H}_{\not n}$, which is the hash of the header excluding the **nonce** fields, i.e. $\mathsf{H}_{\not n} \equiv \mathsf{KEC}(\mathsf{RLP}(\mathsf{H}_{-n}))$, together with the nonce H_n and the dataset $\mathbf{d}$ from appendix F.3.3 as input. The output of MpEthash is the Keccak-256 hash of the concatenation of the seed hash $\mathbf{s}_{\mathrm{h}} \in \mathbb{B}_{512}$ and the compressed mix $\mathbf{m}_{\mathrm{c}} \in \mathbb{B}_{256}$:

$$\mathsf{MpEthash}(\mathsf{H}_{\not n}, \mathsf{H}_n, \mathbf{d}) \equiv \mathsf{KEC}(\mathbf{s}_{\mathrm{h}} \circ \mathbf{m}_{\mathrm{c}}) \tag{317}$$

F.4.1 Multi-point mix

The multi-points $\mathbf{p} \in \mathbb{N}_{64}^{J_{accesses}}$ and multi-point mix $m_{\mathrm{p}} \in \mathbb{N}_{64}$ is calculated from the header $\mathsf{H}_{\not n}$, H_n as follows:

$$\mathbf{p} \equiv \mathbf{p}\left(\mathsf{H}_{\not n}, \mathsf{H}_n\right) \equiv E_{mp-eval}(a, b, c, w, n_{low}, \mathbf{z}) \tag{318}$$

$$m_{\mathrm{p}} \equiv m_{\mathrm{p}}(\mathbf{p}) \equiv E_{\mathrm{FNV}-compress}(\mathbf{p}, J_{accesses}) \tag{319}$$

with arguments and functions described in the rest of this section.

Interpreting $\mathsf{H}_{\not n}$ as a 4-element array of $\mathbb{N}_{64}$ encoded in little-endian, input arguments $a, b, c, w \in \mathbb{N}_{64}, n_{low} \in \mathbb{N}$ are defined as follows:

$$a \equiv E_{remap}(\mathsf{H}_{\not n}[0]) \tag{320}$$

$$b \equiv E_{remap}(\mathsf{H}_{\not n}[1]) \tag{321}$$

$$c \equiv E_{remap}(E_{proper\_c}(a, b, \mathsf{H}_{\not n}[2])) \tag{322}$$

$$w \equiv E_{remap}(\mathsf{H}_{\not n}[3]) \tag{323}$$

$$n_{high} \equiv \lfloor (\mathsf{H}_n \bmod 2^{64}) / J_{warpsize} \rfloor \tag{324}$$

$$n_{low} \equiv \mathsf{H}_n \bmod J_{warpsize} \tag{325}$$

where

$$E_{proper\_power}(h) \equiv \underset{x | h \wedge \gcd(x, J_{mod}-1)=1}{\arg\max} x \tag{326}$$

$$E_{remap}(h) \equiv 11^{E_{proper\_power}((h \bmod (J_{mod}-2))+1)} \bmod J_{mod} \tag{327}$$

$$E_{proper\_c}(a, b, h) \equiv \underset{x \geq h \wedge J_{mod} \nmid b^2-4a \cdot E_{remap}(x)}{\arg\min} x \tag{328}$$

The last argument $\mathbf{z}$ is an array of $J_{pow} = J_{accesses} \times J_{warpsize}$ items drawn from $\mathbb{N}_{32}$. More specifically, $\mathbf{z}[i]$ is defined as:

$$\mathbf{z}[i \cdot J_{warpsize} + j] \equiv E_{\mathrm{Sip},2,j+4}\left(\left(n_{high} \cdot J_{warpsize} + j\right) \bmod 2^{64}\right) \bmod J_{mod}, \quad \forall 0 \leq i < J_{accesses}, 0 \leq j < J_{warpsize} \tag{329}$$

where the $E_{\mathrm{Sip},c,d}$ refers to the SipHash-c-d function with a different key initialization process by $v_i \equiv \mathsf{H}_{\not n}[i]$ ($i \in \{0,1,2,3\}$). See [6] for more details about SipHash function.

The multiple points $\mathbf{p}$ is an array of $J_{accesses}$ many 32-bit integers. Function $E_{mp-eval}$ evaluated the polynomial $E_{polynomial}$ on multiple points $\mathbf{x}[i]$ for $i \in \{0, 1, \ldots, J_{accesses} - 1\}$ to get $\mathbf{p}$.

$$E_{mp-eval}(a, b, c, w, n_{low}, \mathbf{z}) \equiv \left\{E_{polynomial}(\mathbf{x}[0]), E_{polynomial}(\mathbf{x}[1]), \cdots, E_{polynomial}(\mathbf{x}[J_{accesses} - 1])\right\} \tag{330}$$

where $\mathbf{x}$ is an array of words defined as:

$$\mathbf{x}[i] \equiv a \cdot w^{2 \cdot (i \cdot J_{warpsize} + n_{low})} + b \cdot w^{i \cdot J_{warpsize} + n_{low}} + c, \quad \forall \quad 0 \leq i \leq J_{accesses} - 1 \tag{331}$$

and the $E_{polynomial}$ function is a polynomial with coefficients specified by $\mathbf{z}$:

$$E_{polynomial}(x) \equiv \left(\sum_{j=0}^{J_{pow}-1} \mathbf{z}[j] \cdot x^j\right) \bmod J_{mod} \tag{332}$$

The FNV-compress function $E_{\mathrm{FNV}-compress}$ is defined over $(\mathbb{N}_{32})^* \times \mathbb{N} \to \mathbb{N}_{64}$ and used to compress an array of $\mathbb{N}_{32}$ elements into a single $\mathbb{N}_{64}$ element:

$$E_{\mathrm{FNV}-compress}(\mathbf{p}, i) \equiv \begin{cases} \mathbf{0}_{64} & \text{if} \quad i < 1 \\ E_{\mathrm{FNV64}}(E_{\mathrm{FNV}-compress}(\mathbf{p}, i-1), \mathbf{p}[i-1]) & \text{otherwise} \end{cases} \tag{333}$$

where input items in $\mathbb{N}_{32}$ are interpreted as integers in $\mathbb{N}_{64}$, and $E_{\mathrm{FNV64}} : \mathbb{N}_{64} \times \mathbb{N}_{64} \to \mathbb{N}_{64}$ naturally extends E_{FNV} as follows:

$$E_{\mathrm{FNV64}}(\mathbf{x}, \mathbf{y}) \equiv ((\mathbf{x} \times 0\mathrm{x}01000193) \oplus \mathbf{y}) \bmod 2^{64} \tag{334}$$

F.4.2 Half mix

The half mix $\mathbf{s}_\mathrm{h} \in \mathbb{B}_{512} = \mathbb{BY}_{J_{hashbytes}}$ is defined on $\mathsf{H}_{\mathscr{H}}$ and m_p as follows:

$$\mathbf{s}_\mathrm{h} \equiv \mathbf{s}_\mathrm{h}(\mathsf{H}_{\mathscr{H}}, m_\mathrm{p}) \equiv \mathsf{KEC512}\left(\mathsf{H}_{\mathscr{H}} \circ \mathrm{LE}(m_\mathrm{p})\right) \tag{335}$$

where $\mathrm{LE}(m_\mathrm{p})$ returns the little-endian encoding of the compressed multi-point mix m_p.

F.4.3 Compressed mix

The compressed mix $\mathbf{m}_\mathrm{c} \in \mathbb{B}_{256}$ is obtained from the seed hash $\mathbf{s}_\mathrm{h} \in \mathbb{B}_{512} = \mathbb{BY}_{J_{hashbytes}}$, the dataset $\mathbf{d} \in \mathbb{BY}_{d_{size}}$ and the multiple points $\mathbf{p} \in (\mathbb{N}_{64})^{J_{accesses}}$:

$$\mathbf{m}_\mathrm{c} \equiv \mathbf{m}_\mathrm{c}\left(\mathbf{s}_\mathrm{h}, \mathbf{d}, \mathbf{p}\right) \equiv E_{compress}\left(\mathbf{m}_{J_{accesses}}\right) \tag{336}$$

where $\mathbf{m}_{J_{accesses}}$ and $E_{compress}$ are defined as follows.

The initial mix $\mathbf{m}_0$ is an array of n_{mixw} words obtained by replicating the seed hash $\mathbf{s}_\mathrm{h}$ for n_{mixh} times, with n_{mixw}, n_{mixh} defined as:

$$n_{mixw} \equiv \frac{J_{mixbytes}}{J_{wordbytes}} \tag{337}$$

$$n_{mixh} \equiv \frac{J_{mixbytes}}{J_{hashbytes}} \tag{338}$$

Formally, the initial mix $\mathbf{m}_0 \in \mathbb{BY}_{J_{mixbytes}} = \left(\mathbb{BY}_{J_{wordbytes}}\right)^{n_{mixw}}$ is defined as:

$$\mathbf{m}_0 \equiv \underbrace{\mathbf{s}_\mathrm{h} \circ \cdots \circ \mathbf{s}_\mathrm{h}}_{n_{mixh} \text{ many copies of } \mathbf{s}_\mathrm{h}} \tag{339}$$

Every $\mathbf{m}_j$ is updated from $\mathbf{m}_{j-1}$ as follows:

$$\mathbf{m}_j[i] \equiv E_{\mathrm{FNV}}\left(\mathbf{m}_{j-1}[i], \mathbf{d}\left[\left(p_j \cdot n_{mixh} + \lfloor i/n_{hashw} \rfloor\right) \bmod 2^{32}\right]\left[i \bmod n_{hashw}\right]\right), \quad \forall 0 \le j < J_{accesses}, 0 \le i < n_{mixw} \tag{340}$$

$$p_j \equiv E_{\mathrm{FNV}}\left(j \oplus \mathbf{s}_\mathrm{h}[0] \oplus \mathbf{p}[j], \mathbf{m}_{j-1}\left[j \bmod n_{mixw}\right]\right) \bmod \frac{d_{size}}{J_{mixbytes}}, \quad \forall 0 \le j < J_{accesses} \tag{341}$$

where $n_{hashw} \equiv J_{hashbytes}/J_{wordbytes}$. We regard a $J_{hashbytes}$-bit string as an array of n_{hashw} words here.

The $E_{compress}$ function converts $J_{mixbytes}$-byte mix, which is an array of $n_{mixw} = 64$ words, into an 8-word array, with the i-th word defined as follows:

$$\begin{aligned} E_{compress}(\mathbf{m})[i] \equiv E_{\mathrm{FNV}}\big(&E_{\mathrm{FNV}}(E_{\mathrm{FNV}}(E_{\mathrm{FNV}}(\mathbf{m}[4i], \mathbf{m}[4i+1]), \mathbf{m}[4i+2]), \mathbf{m}[4i+3]), \\ &E_{\mathrm{FNV}}(E_{\mathrm{FNV}}(E_{\mathrm{FNV}}(\mathbf{m}[4i+32], \mathbf{m}[4i+33]), \mathbf{m}[4i+34]), \mathbf{m}[4i+35])\big), \quad \forall 0 \le i \le 7 \end{aligned} \tag{342}$$

The array obtained by applying $E_{compress}$ on $\mathbf{m}_{J_{accesses}}$ is indeed the compressed mix $\mathbf{m}_\mathrm{c}$ by eq. (336).

Appendix G. Internal contracts

⟨NOTE: The following fomulars are for Oceanus version.⟩

Conflux introduces internal contracts for specific usage. A high-level description for the internal contracts is given in Section 8. Currently, Conflux has three internal contracts with addresses as follows.

$$a_{\mathsf{admin}} \equiv \texttt{0x0888000000000000000000000000000000000000} \tag{343}$$
$$a_{\mathsf{sponsor}} \equiv \texttt{0x0888000000000000000000000000000000000001} \tag{344}$$
$$a_{\mathsf{stake}} \equiv \texttt{0x0888000000000000000000000000000000000002} \tag{345}$$

When the recipient's address r is one of the internal contracts, Conflux processes $\Xi_{\mathsf{internal}}(\sigma^*, g, I)$ and returns $(\sigma^{**}, g^{**}, A, \mathbf{o})$.

G.1 Interfaces and gas required

In execution of internal contracts, the call data $I_{\mathbf{d}}$ is interpreted as a function call to Solidity interface. Function $V(\sigma, c, I_{\mathbf{d}})$ gives the gas used for internal contract execution by called function and parameters.

Address I_a	Solidity interface	Formal parameters	$V(\sigma, c, I_{\mathbf{d}})$
a_{admin}	`set_admin(address,address)`	$a_0, a_1 \in \mathbb{B}_{160}$	G_{sset}
	`destroy(address)`	$a_0 \in \mathbb{B}_{160}$	G_{sset}
a_{sponsor}	`set_sponsor_for_gas(address,uint256)`	$a_0 \in \mathbb{B}_{160}, n_1 \in \mathbb{N}_{256}$	G_{sset}
	`set_sponsor_for_collateral(address)`	$a_0 \in \mathbb{B}_{160}$	$2 \times G_{sset}$
	`add_privilege(address[])`	$\mathbf{a} \in \mathbb{B}^*_{160}$	$\lvert\mathbf{a}\rvert \times G_{sset}$
	`remove_privilege(address[])`	$\mathbf{a} \in \mathbb{B}^*_{160}$	$\lvert\mathbf{a}\rvert \times G_{sset}$
a_{stake}	`deposit(uint256)`	$n_0 \in \mathbb{N}_{256}$	$2 \times (\lvert\sigma[s]_{deposit}\rvert + 1) \times G_{sset}$
	`withdraw(uint256)`	$n_0 \in \mathbb{N}_{256}$	$2 \times \lvert\sigma[s]_{deposit}\rvert \times G_{sset}$
	`vote_lock(uint256,uint256)`	$n_0, n_1 \in \mathbb{N}_{256}$	$2 \times \lvert 1 + \mathsf{ToList}(\sigma[s]_{vote})\rvert \times G_{sset}$

where $\sigma[s]'_{vote}$ is defined by $\sigma[s]_{vote}$ removing all the key x with $\sigma[s]_{vote}[x] = \sigma[s]_{vote}[x-1]$

G.2 Internal contracts exceptions

The execution of internal contracts may fail in the following cases

- Conflux parses the call data $I_{\mathbf{d}}$ as solidity function call. In case the call data doesn't match any solidity function interfaces list as follows, or the format is incorrect, the execution fails.
- The recipient is inconsistent with the code address or internal contract is called by STATICCALL, i.e., $I_r \neq c$ or $I_w = \bot$.
- The gas g passed in is not enough for internal contract execution, i.e., $g < V(\sigma^*, c, I_{\mathbf{d}})$.
- The staking vote contract a_{stake} forbids value transfer to prevent misusing, i.e., $I_a = a_{\mathsf{stake}} \wedge I_v \neq 0$. [6]
- If other exceptions met during the execution, the execution also fails. See the following for details.

Whenever the execution fails, the return values are set as

$$(\sigma^{**}, g^{**}, A, \mathbf{o}) \equiv (\varnothing, 0, A^0, \varnothing) \tag{346}$$

If the execution is successful, the remained gas and return data are set as

$$(g^{**}, \mathbf{o}) \equiv (g - V(\sigma^*, c, I_{\mathbf{d}}), \varepsilon) \tag{347}$$

The resultant world-state σ^{**} and substate A is computed depending on interfaces.

G.3 Admin Contract

The admin contract with address a_{admin} gives two interfaces.

[6] User don't need to transfer balance to staking vote contract in staking. The transferred value to staking vote contract will lost.

G.3.1 Set administration

For interface `set_admin(address,address)`, let $a_0, a_1 \in \mathbb{B}_{160}$ be the address parameters.

$$A \equiv A^0 \tag{348}$$
$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{349}$$
$$\sigma^{**}[a_0]_a \equiv a_1 \qquad \text{if } \mathsf{Type}_{a_0} = [1000]_2 \wedge I_o = \sigma^*[a_0]_a \tag{350}$$

G.3.2 Destory contract

For interface `destroy(address)`, let $a_0 \in \mathbb{B}_{160}$ be the address parameters. The execution fails if

$$\sigma^*[a_0]_o > 0 \wedge I_o = \sigma^*[a_0]_a \tag{351}$$

If the execution not fails,

$$(\sigma^{**}, A) \equiv \begin{cases} (\sigma^*, A^0) & I_o \neq \sigma^*[a_0]_a \\ \Psi(\sigma^*, A^0) & I_o = \sigma^*[a_0]_a \end{cases} \tag{352}$$

where Ψ is defined in section D.

G.4 Sponsorship Contract

The sponsorship contract with address a_{sponsor} gives four interfaces.

G.4.1 Set sponsor for gas

For interface `set_sponsor_for_gas(address,uint256)`, let $a_0 \in \mathbb{B}_{160}, n_1 \in \mathbb{N}_{256}$ be the address parameters. The execution fails if

$$\begin{aligned} &\sigma^*[a_0] = \emptyset \vee \mathsf{Type}_{a_0} \neq [1000]_2 \vee \sigma^*[a_{\mathsf{sponsor}}]_b < 1000 \times n_1 \vee n_1 \leq \sigma^*[a_0]_p[\mathsf{limit}] \leq \sigma^*[a_0]_p[\mathsf{gas}]_b \\ &\vee (\sigma^*[a_0]_p[\mathsf{gas}]_a \neq I_s \wedge \sigma^*[a_{\mathsf{sponsor}}]_b \leq \sigma^*[a_0]_p[\mathsf{gas}]_b) \end{aligned} \tag{353}$$

If the execution not fails,

$$A \equiv A^0 \tag{354}$$
$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{355}$$
$$\sigma^{**}[a_{\mathsf{sponsor}}]_b \equiv 0 \tag{356}$$
$$\sigma^{**}[a_0]_p[\mathsf{limit}] \equiv n_1 \tag{357}$$
$$\sigma^{**}[a_0]_p[\mathsf{gas}]_a \equiv a_0 \tag{358}$$
$$\sigma^{**}[a_0]_p[\mathsf{gas}]_b \equiv \begin{cases} \sigma^*[a_{\mathsf{sponsor}}]_b + \sigma^*[a_0]_p[\mathsf{gas}]_b & p = I_s \\ \sigma^*[a_{\mathsf{sponsor}}]_b & p \neq I_s \end{cases} \tag{359}$$
$$\sigma^{**}[p]_b \equiv \begin{cases} \sigma^*[p]_b & p = I_s \\ \sigma^*[p]_b + \sigma^*[a_0]_p[\mathsf{gas}]_b & p \neq I_s \end{cases} \tag{360}$$
$$\text{where:} \tag{361}$$
$$p \equiv \sigma^*[a_0]_p[\mathsf{gas}]_a \tag{362}$$

G.4.2 Set sponsor for collateral

For interface `set_sponsor_for_collateral(address)`, let $a_0 \in \mathbb{B}_{160}$ be the address parameter. The execution fails if

$$\begin{aligned} &\sigma^*[a_0] = \emptyset \vee \mathsf{Type}_{a_0} \neq [1000]_2 \vee \sigma^*[a_{\mathsf{sponsor}}]_b = 0 \\ &\vee (\sigma^*[a_0]_p[\mathsf{col}]_a \neq I_s \wedge \sigma^*[a_{\mathsf{sponsor}}]_b \leq \sigma^*[a_0]_p[\mathsf{col}]_b) \end{aligned} \tag{363}$$

If the execution not fails,

$$A \equiv A^0 \tag{364}$$
$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{365}$$
$$\sigma^{**}[a_{\mathsf{sponsor}}]_b \equiv 0 \tag{366}$$
$$\sigma^{**}[a_0]_p[\mathsf{col}]_a \equiv a_0 \tag{367}$$
$$\sigma^{**}[a_0]_p[\mathsf{col}]_b \equiv \begin{cases} \sigma^*[a_{\mathsf{sponsor}}]_b + \sigma^*[a_0]_p[\mathsf{col}]_b & p = I_s \\ \sigma^*[a_{\mathsf{sponsor}}]_b & p \neq I_s \end{cases} \tag{368}$$
$$\sigma^{**}[p]_b \equiv \begin{cases} \sigma^*[p]_b & p = I_s \\ \sigma^*[p]_b + \sigma^*[a_0]_p[\mathsf{col}]_b & p \neq I_s \end{cases} \tag{369}$$
$$\text{where:} \tag{370}$$
$$p \equiv \sigma^*[a_0]_p[\mathsf{col}]_a \tag{371}$$

G.4.3 Add addresses to whitelist

For interface `add_privilege(address[])`, let $\mathbf{a} \in \mathbb{B}^*_{160}$ be the address list parameter. The execution fails if

$$\mathsf{Type}_{I_s} \neq [1000]_2. \tag{372}$$

If the execution not fails,

$$(\sigma^{**}, A) \equiv \left(\sigma^{(n)}, A^0\right) \tag{373}$$
$$\text{where:} \tag{374}$$
$$n \equiv |\mathbf{a}| \tag{375}$$
$$\sigma^{(0)} \equiv \sigma^* \tag{376}$$
$$\forall j \in [n],\ \sigma^{(j)} \equiv \Phi(\sigma^{(j-1)}, a_{\mathsf{sponsor}}, I_s \cdot \mathbf{a}[j-1], 1, I_i) \tag{377}$$

Function Φ is defined in Section 7.1.

G.4.4 Remove addresses to whitelist

For interface `remove_privilege(address[])`, let $\mathbf{a} \in \mathbb{B}^*_{160}$ be the address list parameter. The execution fails if

$$\mathsf{Type}_{I_s} \neq [1000]_2. \tag{378}$$

If the execution not fails,

$$(\sigma^{**}, A) \equiv \left(\sigma^{(n)}, A^0\right) \tag{379}$$
$$\text{where:} \tag{380}$$
$$n \equiv |\mathbf{a}| \tag{381}$$
$$\sigma^{(0)} \equiv \sigma^* \tag{382}$$
$$\forall j \in [n],\ \sigma^{(j)} \equiv \Phi(\sigma^{(j-1)}, a_{\mathsf{sponsor}}, I_s \cdot \mathbf{a}[j-1], 0, I_i) \tag{383}$$

G.5 Staking vote contract

The staking vote contract with address a_{stake} gives three interfaces:

G.5.1 Staking

For interface `deposit(uint256)`, let $n_0 \in \mathbb{N}_{256}$ be the integer parameter. The execution fails if

$$n_0 < 10^{18} \ \vee\ \sigma^*[I_s]_b < n_0 \tag{384}$$

If the execution not fails,

$$A \equiv A^0 \tag{385}$$
$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{386}$$
$$\sigma^{**}[I_s]_b \equiv \sigma^*[I_s]_b - n_0 \tag{387}$$
$$\sigma^{**}[I_s]_t \equiv \sigma^*[I_s]_t + n_0 \tag{388}$$
$$\sigma^{**}[I_s]_{\mathbf{d}} \equiv \sigma^*[I_s]_{\mathbf{d}} \cdot (n_0, |I_{\mathbf{L}}|, \sigma^*[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_1]_v) \tag{389}$$
$$\sigma^{**}[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_2]_v \equiv \sigma^*[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_2]_v + n_0 \tag{390}$$
$$\text{where:} \tag{391}$$
$$k_1 \equiv [\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{392}$$
$$k_2 \equiv [\mathsf{total\_staking\_tokens}]_{\mathsf{ch}} \tag{393}$$

G.5.2 Lock staking to obtain vote power

For interface `vote_lock(uint256,uint256)`, let $n_0, n_1 \in \mathbb{N}_{256}$ be the integer parameters. The execution fails if

$$n_1 \leq |I_{\mathbf{L}}| \ \vee \ \sigma^*[I_s]_t < n_0 \tag{394}$$

If the execution not fails,

$$A \equiv A^0 \tag{395}$$
$$\sigma^{**} \equiv \sigma^* \qquad \text{except:} \tag{396}$$
$$\forall x \leq |I_{\mathbf{L}}|, \sigma^{**}[I_s]_{\mathbf{v}}[x] \equiv \sigma^*[I_s]_{\mathbf{v}}[|I_{\mathbf{L}}|+1] \tag{397}$$
$$\forall |I_{\mathbf{L}}| < x \leq n_1, \sigma^{**}[I_s]_{\mathbf{v}}[x] \equiv \max\{\sigma^*[I_s]_{\mathbf{v}}[x], n_0\} \tag{398}$$

G.5.3 Withdraw

For interface `withdraw(uint256)`, let $n_0 \in \mathbb{N}_{256}$ be the integer parameter. The execution fails if

$$n_0 > \sigma[I_s]_t - \sigma[I_s]_{\mathbf{v}}[|I_{\mathbf{L}}|+1] \tag{399}$$

If the execution not fails,

$$A \equiv A^0 \tag{400}$$
$$\sigma^1 \equiv \sigma^* \qquad \text{except:} \tag{401}$$
$$\forall x \leq |I_{\mathbf{L}}|, \sigma^1[I_s]_{\mathbf{v}}[x] \equiv \sigma^*[I_s]_{\mathbf{v}}[|I_{\mathbf{L}}|+1] \tag{402}$$
$$\sigma^1[I_s]_t \equiv \sigma^*[I_s]_t - n_0 \tag{403}$$
$$\forall i < |\sigma^*[I_s]_{\mathbf{d}}|, \sigma^1[I_s]_{\mathbf{d}}[i][\mathsf{amt}] \equiv \sigma^*[I_s]_{\mathbf{d}}[i][\mathsf{amt}] - z_i \tag{404}$$
$$\sigma^1[I_s]_b \equiv \sigma^*[I_s]_b + n_0 + q \tag{405}$$
$$\sigma^1[I_s]_r \equiv \sigma^*[I_s]_r + q \tag{406}$$
$$\sigma^1[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_2]_v \equiv \sigma^*[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_2]_v - n_0 \tag{407}$$
$$\sigma^1[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_3]_v \equiv \sigma^*[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_3]_v + q \tag{408}$$
$$\sigma^{**} \equiv \sigma^1 \qquad \text{except:} \tag{409}$$
$$\sigma^{**}[I_s]_{\mathbf{d}} \equiv \text{Remove elements } e \text{ from } \sigma^1[I_s]_{\mathbf{d}} \text{ with } e[\mathsf{amt}] = 0 \tag{410}$$
$$\text{where:} \tag{411}$$
$$y_i \equiv \sum_{j=0}^{i-1} \sigma^*[I_s]_{\mathbf{d}}[j][\mathsf{amt}] \tag{412}$$
$$z_i \equiv \max\{0, \min\{n_0 - y_i, \sigma^*[I_s]_{\mathbf{d}}[i][\mathsf{amt}]\}\} \tag{413}$$
$$q \equiv \sum_{i=0}^{|\sigma^*[I_s]_{\mathbf{d}}|-1} \left\lfloor \frac{z_i \times \sigma^*[a_{\mathsf{sponsor}}]_{\mathbf{s}}[k_1]}{\sigma^*[I_s]_{\mathbf{d}}[i][\mathsf{acclR}]} \right\rfloor \tag{414}$$
$$k_1 \equiv [\mathsf{accumulate\_interest\_rate}]_{\mathsf{ch}} \tag{415}$$
$$k_2 \equiv [\mathsf{total\_staking\_tokens}]_{\mathsf{ch}} \tag{416}$$
$$k_3 \equiv [\mathsf{total\_issued\_tokens}]_{\mathsf{ch}} \tag{417}$$