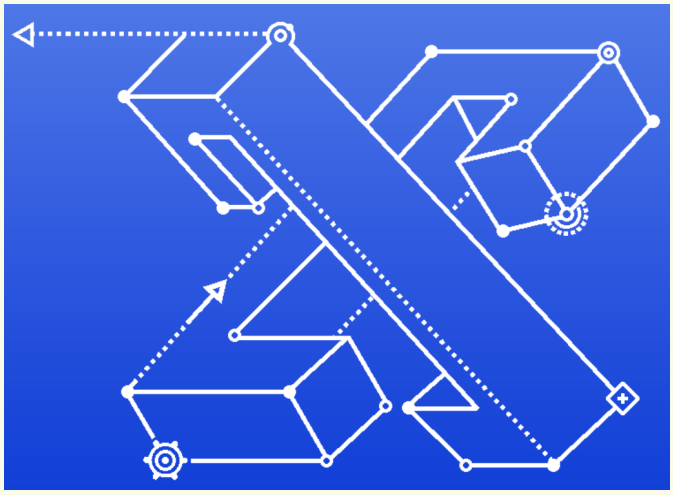
[Testnet v0.6]
April 30, 2019

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 $\mathbb{N}_n \equiv \{0,1,\dots,2^n-1\}$ be the set of non-negative integers smaller than 2^n.
- 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.
  - 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.
- 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 $\boldsymbol{\mu}_\mathbf{m}[0\dots31]$ denotes the first 32 items of the machine's memory.
  - The global state $\boldsymbol{\sigma}$ is interpreted as a sequence of accounts, where each account is a tuple. Thus the square bracket after $\boldsymbol{\sigma}$ refers to an individual (or a list of) accounts.
- 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_{SSTORE} is the cost function for the SSTORE 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. $F^*\big((x_1,x_2,\dots)\big) \equiv (F(x_1),F(x_2),\dots)$.
- The suprescript 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 $i \geq 2$.

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, $\mathsf{SIBLING}(\mathsf{B}) \equiv \{\mathsf{B}' | \mathsf{P}(\mathsf{B}') = \mathsf{P}(\mathsf{B}) \wedge \mathsf{B}' \neq \mathsf{B}\}$.
- 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.
- FUTURE: the *future function* takes a graph $\mathbf{G}$ and a block B as inputs and outputs all blocks in the "future set of B". Formally, $\mathsf{FUTURE}(\mathbf{G},\mathsf{B}) \equiv \{\mathsf{B}' \in \mathbf{G} | \mathsf{B} \in \mathsf{PAST}(\mathsf{B}')\}$.
- 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 4.2.3.
- 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].
- 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}.
- PoW: this is the proof-of-work function, which takes a block header as input and returns a scalar in $\mathbb{B}_{256}$.

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

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^{18}	Conflux (CFX)

3. Basic Components

In an overview, the Conflux global 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 state. The account address α_{addr} is a 160-bit identifier, and the account state α_{state} is a serialized sequence in an RLP structure (c.f. [3]). Furthermore we note that each account α is associated with a pair of public key and private key $(\alpha_{pubkey}, \alpha_{prikey})$. The account address is essentially a digest of the associated public key, i.e. $\alpha_{addr} \equiv \mathsf{KEC}(\alpha_{pubkey})[0\ldots159]$.

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.

An account state $\alpha_{state} \equiv (\alpha_n, \alpha_b, \alpha_s, \alpha_c)$ consists of the following four fields:

- **nonce:** A scalar counter recording the number of previous activities initiated by this account. Formally denoted by α_n. 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 α_b.
- **storageRoot:** A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account. It encodes a key/value database into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. This root node hash is formally denoted by α_s.
- **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 α_c, which satisfies $\alpha_c = \mathsf{KEC}(\mathbf{p})$ when the stored code is $\mathbf{p}$.

Typically we refer to the underlying set of key/value pairs stored in the trie rather than the root hash α_s. For convenience we define the following equivalence:

$$\mathsf{TRIE}(L_I^*(\sigma[a]_s)) \equiv \sigma[a]_s \tag{1}$$

where the collapse function L_I^* is defined as the element-wise transformation of the base function L_I, which applies on a single pair of key/value $(k,v) \in \mathbb{B}_{256} \times \mathbb{N}$ as follows:

$$L_I((k,v)) \equiv (\mathsf{KEC}(k), \mathsf{RLP}(v)) \tag{2}$$

If the **codeHash** field of an account α is the hash of the empty string, i.e. $\alpha_c = \mathsf{KEC}(())$, then α is a *simple account*, also called *"non-contract" account*.

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. $\sigma[a] \equiv (\alpha_n, \alpha_b, \alpha_s, \alpha_c)$. We denote by $\sigma[a] \equiv \emptyset$ if the account with address a is never initialized.

Given a world state σ, an account is *empty* in this state if it has zero nonce, zero balance, and no code.

$$\mathsf{EMPTY}(\sigma, a) \equiv \left[\sigma[a]_c = \mathsf{KEC}(()) \wedge \sigma[a]_n = 0 \wedge \sigma[a]_b = 0\right] \tag{3}$$

An account is *dead* if its account state is non-existent or empty.

$$\mathsf{DEAD}(\sigma, a) \equiv [\sigma[a] = \emptyset \vee \mathsf{EMPTY}(\sigma, a)] \tag{4}$$

We also define the account validity function v as follows:

$$v(\alpha) \equiv [\alpha_n \in \mathbb{N}_{256} \wedge \alpha_b \in \mathbb{N}_{256} \wedge \alpha_s \in \mathbb{B}_{256} \wedge \alpha_c \in \mathbb{B}_{256}] \tag{5}$$

For every legal world-state σ and for every address $a \in \mathbb{B}_{160}$, the account with address a is either unused or valid, i.e.

$$\forall a \in \mathbb{B}_{160} : (\sigma[a] = \emptyset) \vee (v(\sigma[a])) \tag{6}$$

Thus we define the *world-state collapse function* L_S that translates the world-state σ into account states:

$$L_S(\sigma) \equiv \left\{ \left(a, \mathsf{RLP}(\sigma[a]_n, \sigma[a]_b, \sigma[a]_s, \sigma[a]_c)\right) \;\middle|\; a \in \mathbb{B}_{160} \wedge \sigma[a] \neq \emptyset \right\} \tag{7}$$

This function L_S and the trie function TRIE are used in conjunction to provide a short hash digest of the Conflux world-state.

3.2 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**.
- **to:** A variable size field indicating the recipient of this transaction, formally denoted by T_t. There is an extra sub-field **create** indicating whether T is a creation transaction, which we make it implicit for notation convenience. Thus, for action transactions we interpret T_t as the 160-bit address of the recipient; otherwise in case of a creation transaction, the recipient is indeed the newly created contract and we interpret T_t as the only element in $\mathbb{B}_0$ and write $\mathsf{T}_t = \emptyset$.
- **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_t or the newly created contract. Formally denoted by $\mathsf{T}_v \in \mathbb{N}_{256}$.
- **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 S 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:

$$L_{\mathsf{T}}(\mathsf{T}) \equiv \begin{cases} (\mathsf{T}_n, \mathsf{T}_p, \mathsf{T}_g, \mathsf{T}_t, \mathsf{T}_v, \mathsf{T}_{\mathbf{i}}, \mathsf{T}_w, \mathsf{T}_r, \mathsf{T}_s) & \text{if } \mathsf{T}_t = \emptyset \\ (\mathsf{T}_n, \mathsf{T}_p, \mathsf{T}_g, \mathsf{T}_t, \mathsf{T}_v, \mathsf{T}_{\mathbf{d}}, \mathsf{T}_w, \mathsf{T}_r, \mathsf{T}_s) & \text{otherwise} \end{cases} \tag{8}$$

3.3 Blocks

The Conflux blockchain organizes all on-chain information in blocks. Every Conflux block B consists of three parts: a block header H, a list of comprised transactions Ts, and a list of other unreferenced block headers U (a.k.a. *ommer* blocks or simply *ommers*[2]).

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 $\mathsf{H}_p \in \mathbb{B}_{256}$.
- **ommersHash:** The Keccak 256-bit hash of the ommers list part of this block, formally denoted by $\mathsf{H}_o \in \mathbb{B}_{256}$.
- **beneficiary:** The recipient's 160-bit address to receive all rewards caused by successfully mining this block, formally denoted by $\mathsf{H}_c \in \mathbb{B}_{160}$.
- **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 $\mathsf{H}_t \in \mathbb{B}_{256}$.
- **stateRoot:** The Keccak 256-bit hash of the root node of the state trie 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.
- **receiptsRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction executed when updating **stateRoot** of the block, formally $\mathsf{H}_e \in \mathbb{B}_{256}$.
- **logsBloom:** The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transaction list, formally H_b.
- **difficulty:** A scalar value specifying the target difficulty level of this block. This is calculated from the previous block's difficulty level and the timestamp. formally $\mathsf{H}_d \in \mathbb{N}_{64}$.
- **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 $\mathsf{H}_h \in \mathbb{N}_{32}$. The genesis block has a height of zero.
- **gasLimit:** A scalar value equal to the current limit of gas expenditure per block. It is formally denoted by $\mathsf{H}_\ell \in \mathbb{N}_{64}$.
- **timestamp:** A scalar value equal to the reasonable output of Unix's time() at this block's inception. Formally denoted by $\mathsf{H}_s \in \mathbb{B}_{64}$.
- **extraData:** An arbitrary byte array containing ≤ 32 bytes data relevant to this block. Formally denoted by $\mathsf{H}_x \in \mathbb{BY}_{32}$.
- **nonce:** A 64-bit hash which, combined with the mix-hash, proves that a sufficient amount of computation has been carried out on this block; formally $\mathsf{H}_n \in \mathbb{B}_{64}$.

The other two parts of the block B are simply a list of transactions and a list of ommer block headers. Therefore the block B can be represented as follows:

$$\mathsf{B} \equiv (\mathsf{B}_\mathsf{H}, \mathsf{B}_\mathsf{Ts}, \mathsf{B}_\mathsf{U}) \tag{9}$$

3.3.1 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 **stateRoot** 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 $R \equiv (R_u, R_b, R_\mathbf{l}, R_z)$ is a tuple consisting of four fields:

- $R_u \in \mathbb{N}$ is the cumulative gas used as of immediately after T has been processed;
- $R_\mathbf{l}$ 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.

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 $O_\mathbf{d} \in \mathbb{B}^*$:

$$O \equiv (O_a, O_\mathbf{t}, O_\mathbf{d}) \tag{10}$$

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

$$M(O) \equiv \bigvee_{x \in \{O_a\} \cup O_\mathbf{t}} (M_{3:2048}(x)) \tag{11}$$

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].

[2] *Ommer* is a non-standard gender-neutral term to mean "sibling of parent"; see https://nonbinary.miraheze.org/wiki/Gender_neutral_language_in_English. However, we remark that an "ommer block" is not necessarily a sibling of the parent block – it can be any unreferenced block at any height (may even be higher than the currenct block).

3.3.2 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. (8)), 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:

$$L_H(\mathsf{H}) \equiv (\mathsf{H}_p, \mathsf{H}_o, \mathsf{H}_c, \mathsf{H}_r, \mathsf{H}_t, \mathsf{H}_e, \mathsf{H}_b, \mathsf{H}_d, \mathsf{H}_h, \mathsf{H}_\ell, \mathsf{H}_g, \mathsf{H}_s, \mathsf{H}_x, \mathsf{H}_n) \tag{12}$$

$$L_B(\mathsf{B}) \equiv (L_H(\mathsf{B}_\mathsf{H}), L^*_\mathsf{T}(\mathsf{B}_\mathsf{Ts}), L^*_H(\mathsf{B}_\mathsf{U})) \tag{13}$$

The component types are defined thus:

$$\begin{aligned} & \mathsf{H}_p \in \mathbb{B}_{256} \quad \wedge \quad \mathsf{H}_o \in \mathbb{B}_{256} \quad \wedge \quad \mathsf{H}_c \in \mathbb{B}_{160} \quad \wedge \quad \mathsf{H}_r \in \mathbb{B}_{256} \quad \wedge \quad \mathsf{H}_t \in \mathbb{B}_{256} \\ \wedge \quad & \mathsf{H}_e \in \mathbb{B}_{256} \quad \wedge \quad \mathsf{H}_b \in \mathbb{BY}_{256} \quad \wedge \quad \mathsf{H}_d \in \mathbb{N}_{64} \quad \wedge \quad \mathsf{H}_h \in \mathbb{N}_{64} \quad \wedge \quad \mathsf{H}_\ell \in \mathbb{N}_{64} \\ \wedge \quad & \mathsf{H}_s \in \mathbb{N}_{256} \quad \wedge \quad \mathsf{H}_x \in \mathbb{BY}_{\leq 32} \quad \wedge \quad \mathsf{H}_n \in \mathbb{B}_{64} \end{aligned} \tag{14}$$

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.

3.3.3 Block Header Validity

Given a block B, we decide whether the header of B is valid by checking the following fields of B_H and comparing to $\mathsf{H}(\mathsf{P}(\mathsf{B}))$ or $\mathsf{PAST}(\mathsf{B})$ if 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 5000;
- the target difficulty is properly set according to Section 6.1;
- the proof-of-work satisfies the target difficulty;
- the **extraData** is no more than 32 bytes;
- the parent is chosen properly from the past view of B following the GHOST rule [1];
- the deferred state root **stateRoot** must be correct. More specifically, it commits to the state right after executing transactions in $\mathsf{EPOCH}\left(\mathsf{P}^{(5)}(\mathsf{B})\right)$.

Formally, the block B has a valid header if and only if:

$$\mathsf{B}_{\mathsf{H}_h} = \mathsf{P}(\mathsf{B})_{\mathsf{H}_h} + 1 \tag{15}$$

$$\wedge \quad \mathsf{B}_{\mathsf{H}_s} > \mathsf{P}(\mathsf{B})_{\mathsf{H}_s} \tag{16}$$

$$\wedge \quad \mathsf{B}_{\mathsf{H}_l} < \left(1 + \frac{1}{1024}\right) \mathsf{P}(\mathsf{B})_{\mathsf{H}_l} \quad \wedge \quad \mathsf{B}_{\mathsf{H}_l} > \left(1 - \frac{1}{1024}\right) \mathsf{P}(\mathsf{B})_{\mathsf{H}_l} \quad \wedge \quad \mathsf{B}_{\mathsf{H}_l} \geq 5000 \tag{17}$$

$$\wedge \quad \mathsf{PoW}(\mathsf{B}_\mathsf{H}) \leq \frac{2^{256}}{\mathsf{B}_{\mathsf{H}_d}} \tag{18}$$

$$\wedge \quad \|\mathsf{B}_{\mathsf{H}_x}\| \leq 32 \tag{19}$$

$$\wedge \quad \mathsf{B}_{\mathsf{H}_r} = \mathsf{TRIE}\left(L_S\left(\mathscr{C}\left(\sigma, \mathsf{EPOCH}\left(\mathsf{P}^{(5)}(\mathsf{B})\right)\right)\right)\right) \tag{20}$$

$$\wedge \quad \mathsf{B}_{\mathsf{H}_d} \text{ is legitimate according to the difficulty adjusting function} \tag{21}$$

$$\wedge \quad \mathsf{P}(\mathsf{B}) \text{ is legitimate according to GHOST rule in } \mathsf{PAST}(\mathsf{B}) \tag{22}$$

In (20), σ refers to the base state just before executing $\mathsf{EPOCH}\left(\mathsf{P}^{(5)}(\mathsf{B})\right)$. In particular, σ is exactly the final state after executing $\mathsf{P}^{(6)}(\mathsf{B})$. Furthermore, recalling that L_S is the world-state collapse function, the state root of σ is stored in the header of $\mathsf{P}(\mathsf{B})$ and it satisfies the following:

$$\mathsf{TRIE}\left(L_S(\sigma)\right) = \mathsf{P}(\mathsf{B})_{\mathsf{H}_r} \tag{23}$$

3.3.4 Partially (In)Valid Blocks

We consider a block header $\mathsf{H} = \mathsf{H}(\mathsf{B})$ *partially valid* if it passes all the assertions as in Section 3.3.3 except for the following two:

- the deferred state root H_r is incorrect;

- the parent reference $\mathsf{P}(\mathsf{B})$ is not chosen by the GHOST rule in $\mathsf{PAST}(\mathsf{B})$.

A partially valid block has no reward, zero weight and cannot be chosen as a pivot block. Thus, such a block will not contribute to the security of Conflux consensus.

However, 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.

3.3.5 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 conditions:

$$\mathsf{H}_o \equiv \mathsf{KEC}\left(\mathsf{RLP}(L_H^*(\mathsf{B}_\mathsf{U}))\right) \tag{24}$$

$$\wedge \quad \mathsf{H}_t \equiv \mathsf{TRIE}\left(\forall i < ||\mathsf{B}_\mathsf{Ts}||, i \in \mathbb{N} : (\mathsf{RLP}(i), \mathsf{RLP}(L_\mathsf{T}(\mathsf{B}_\mathsf{Ts}[i])))\right) \tag{25}$$

$$\wedge \quad \mathsf{H}_e \equiv \mathsf{TRIE}\left(\forall i < ||\mathsf{B}_\mathbf{R}||, i \in \mathbb{N} : (\mathsf{RLP}(i), \mathsf{RLP}(L_R(\mathsf{B}_\mathbf{R}[i])))\right) \tag{26}$$

$$\wedge \quad \mathsf{H}_b \equiv \vee_{r \in \mathsf{B}_\mathbf{R}}\left(r_b\right) \tag{27}$$

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.

Topological Consistency. From the parent and ommer references of a block B, we can construct a DAG with B being a leaf block. The referenced blocks are *topologically consistent* if there is no clear chronological order between any two of them, i.e. they appear in each other's anti-cone zones. This is required since otherwise a valid block B should only reference the one appears later, which would already reference the earlier block directly or indirectly.

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 Conflux consensus rules in [4]. 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 ommer 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 three outcomes:

- **accept:** the block is valid 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, the validation 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 3.3.3 and 3.3.4. 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 $\mathsf{B}_{\mathsf{H}_d}$. The gas limit $\mathsf{B}_{\mathsf{H}_\ell} \in \left(1 \pm \frac{1}{1024}\right) \times \mathsf{P}(\mathsf{B})_{\mathsf{H}_\ell}$ and $\mathsf{B}_{\mathsf{H}_\ell} \geq 5000$ is also checked here.

   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. **Ommer Validation.** The validation of ommer 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 referenced 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. **Internal Consistency.** This step asserts that B is self-consistent. More specifically:
   - the block header B_H is formally consistent with content in $\mathsf{B}_\mathsf{Ts}, \mathsf{B}_\mathsf{U}$, i.e. B_H is well-formed following Section 3.3.5;
   - 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 $\mathsf{S}(\mathsf{T})$;
   - the total gas consumption does not exceed the block gas limit, i.e. $\sum_{\mathsf{T}\in\mathsf{B}_\mathsf{Ts}} \mathsf{T}_g \leq \mathsf{B}_{\mathsf{H}_\ell}$.

4. 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) ommer 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 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 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 selects 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, Conflux selects the pivot chain based on the GHOST rule [1] (a simplified variant of the GHOST rule is used in Ethereum). The Conflux selection algorithm starts from the genesis block. At each step, it computes the accumulated *total difficulty* of each child subtree in the parental tree and advances to the child block whose subtree has the largest total difficulty, until it reaches a leaf block.

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

$$\mathsf{B}_t \equiv \mathsf{B}_d + \sum_{\mathsf{B}':\mathsf{P}(\mathsf{B}')=\mathsf{B}} \mathsf{B}'_t \tag{28}$$

where $\mathsf{B}_d = \mathsf{H}(\mathsf{B})_d$ 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 difficulties (and hence the whole pivot chain) can be computed from all block headers, since a block header contains the block difficulty and its parent block, which is all we need to compute the total difficulty recursively.

The advantage of the GHOST rule is that it guarantees the irreversibility of the selected pivot chain even when facing forks of honest nodes due to network delays, since the blocks in the forks also contribute to the safety of the pivot chain (c.f. [1]).

For example, consider the local view as in Figure 1 and for simplicity suppose that all blocks have equal difficulty. 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 and 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.

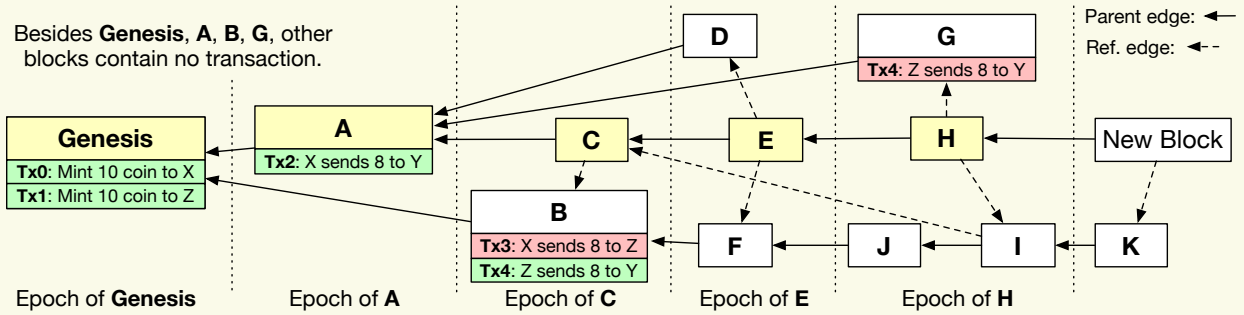


Figure 1. An example local Tree-Graph state to illustrate the consensus algorithm of Conflux. The yellow blocks are on the pivot chain in the Tree-Graph. Each block on the pivot chain forms a new epoch to partition blocks in the Tree-Graph.

4.2.2 Epoch

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

- Every pivot block B is at epoch $H(B)_h$, or simply the *epoch of* B, which is denoted by $\text{EPOCH}(B)$ 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 $B' \in B_U$ is at the epoch of B; then every block B'' referenced by B', i.e. $B'' \in B'_U \cup \{P(B')\}$, 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 contains all blocks that are potentially produced after $P(B)$ but before B at the local view of B.

For example, in Figure 1, each of the pivot blocks Genesis, A, C, E, and H corresponds to one epoch. The block J belongs to the epoch of H but not E because J is reachable from H but not reachable from the previous pivot block E.

4.2.3 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 ommer 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 : The local Tree-Graph $\mathbf{G} = \langle \mathbf{B}, P, E \rangle$ and a block $B \in \mathbf{B}$
Output : A list of blocks $\mathbf{L} = B_1 \circ B_2 \circ \ldots \circ B_n$, where $B_1 = G$ and $\forall 1 \le i \le n, B_i \in \mathbf{B}$

$B' \longleftarrow P(B)$
if $B' = \perp$ **then**
  return B
$\mathbf{L} \longleftarrow \text{ConfluxOrder}(\mathbf{G}, B')$
$\mathbf{L}' \longleftarrow$ An empty list
$\Delta \longleftarrow \text{PAST}(\mathbf{G}, B) - \text{PAST}(\mathbf{G}, B')$
while $\Delta \neq \emptyset$ **do**
  $\Delta' \longleftarrow \{\tilde{B} \mid |\text{FUTURE}(\mathbf{G}, \tilde{B}) \cap \Delta| = 0\}$
  Let B' be the block with maximum $\text{Hash}(B')$ in Δ'
  $\mathbf{L}' \longleftarrow B' \circ \mathbf{L}'$
  $\Delta \longleftarrow \Delta - B'$
$\mathbf{L} \longleftarrow \mathbf{L} \circ \mathbf{L}'$
return $\mathbf{L}$

Figure 2. The Definition of ConfluxOrder().

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.

Transaction Total Order: Conflux first sorts transactions based on the total orders of their enclosing blocks. In case two transactions belong to the same block, Conflux sorts them based on their appearance order in the block. Conflux checks the conflicts of the transactions at the same time when deriving the orders. If two transactions are conflicting with each other, Conflux will discard the second one. If one transaction appears in multiple blocks, Conflux will only keep the first appearance 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 Tx2, and discards the second Tx4 because it is redundant.

4.3 Finalization

In this part we discuss when a block or a transaction in Conflux is considered irreversible, or "final". 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 difficulty 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 as 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[3]. 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 final as follows:

1. Estimate or make assumptions about 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). The user needs to choose an upper bound for the maximum delay d.
2. For each block A in $\mathsf{CHAIN}(\mathsf{B})$, collect the following numbers:
   - w_0: the total difficulty of all the seen blocks;
   - w_1: the total difficulty of subtree rooted at A;
   - w_2: the maximum subtree difficulty among all the subtrees rooted at $\mathsf{SIBLING}(\mathsf{A})$;
   - w_3: the total difficulty of all the blocks in $\mathsf{PAST}(\mathsf{A})$;
   - w_4: the total difficulty of received blocks in the past $2d$ time;
   - $\mathbf{d}$: the largest target difficulty in the last one day.
3. Upper bound the risk that B is reverted and kicked off from the pivot chain:
   - Let n, m and $\zeta_k(m)$ be defined as follows:

$$n \equiv \lceil (w_1 - w_2 - w_4)/\mathbf{d} \rceil,\ m \equiv \lfloor (w_0 - w_3)/\mathbf{d} \rfloor \tag{29}$$

$$\zeta_k(m) \equiv \binom{m+k-1}{k} \cdot \frac{q^k}{(1+q)^{m+k}} \tag{30}$$

   - The probability that block A being kicked off from the pivot chain is bounded as follows:

$$\mathsf{Risk}(\mathsf{A}) \equiv \sum_{k=0}^{+\infty} \zeta_k(m) \cdot \min\{q^{n-k}, 1\} \tag{31}$$

   - The risk that B (and hence B') is reverted is bounded by the maximum risk of all blocks in $\mathsf{CHAIN}(\mathsf{B})$. Formally,

$$\text{The risk of } \mathsf{B} \text{ being reverted} \leq \max_{\mathsf{A} \in \mathsf{CHAIN}(\mathsf{B})} \mathsf{Risk}(\mathsf{A}) \tag{32}$$

4. B is finalized when the risk of B being reverted becomes tolerable, i.e. smaller than some tiny constant.

Since evaluating $\mathsf{Risk}(\mathsf{A})$ may be costly, we provide the following simple (but not tight) formulas for risk estimation, where n, m are defined as in (29). These estimations are conservative and hence result in a longer confirmation time. Users, especially those who are also developers, are free to make more accurate estimation and design sophisticated strategies to achieve a better balance between confirmation time and security.

[3] 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.

Adversary power \ Error tolerance	$\mathsf{Risk}(\mathsf{A}) < 10^{-4}$	$\mathsf{Risk}(\mathsf{A}) < 10^{-6}$
$q \leq 0.25$ (i.e. 20% adversary)	$n > \min\left\{\frac{2m}{5}+13,\ \frac{2m}{7}+36\right\}$	$n > \min\left\{\frac{2m}{5}+19,\ \frac{2m}{7}+57\right\}$
$q \leq 0.5$ (i.e. 33.3% adversary)	$n > \min\left\{\frac{2m}{3}+23,\ \frac{6m}{11}+64\right\}$	$n > \min\left\{\frac{2m}{3}+36,\ \frac{6m}{11}+103\right\}$

5. Transaction Processing

Conflux implements the same virtual machine as Ethereum [3]. Roughly speaking, after determining the total order of transactions and removing invalid transactions, the remaining valid transactions are executed on the EVM as if they are packed into sequential blocks on an Ethereum-like chain. In what follows we focus on the Conflux specific designs in the execution.

5.1 Gas and Payment

As defined in Section 3.2 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 is considered invalid if the sender's account balance cannot afford such a purchase. Like in Ethereum, gas does not exist outside the execution of transactions.

However, note that in the current version of Conflux, the unused gas at the end of the transaction execution is *not refundable* – it is the sender's responsibility to estimate the amount of gas to be consumed and set an appropriate gas limit. This design makes it obvious for miners to calculate and compare the transaction fee when packing transactions into a new block, without actually executing any transactions. Indeed, the execution of transactions is deferred to later blocks, which is particularly important for efficiency of executing transactions in concurrent blocks.

The Conflux used to purchase gas 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.

5.2 Transaction Validation

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

1. The transaction nonce is valid, i.e. $\mathsf{T}_n = \sigma\left[\mathsf{S}(\mathsf{T})\right]_n$ where σ is the current world state.
2. The gas limit T_g is no smaller than the intrinsic gas g_0 used by the transaction.
3. The sender account balance contains at least the cost required in up-front payment, i.e. $\sigma\left[\mathsf{S}(\mathsf{T})\right]_b \geq \mathsf{T}_p\mathsf{T}_g + \mathsf{T}_v$.
   Note: in case T passes all previous checks but fails the last one with $\sigma\left[\mathsf{S}(\mathsf{T})\right]_b < \mathsf{T}_p\mathsf{T}_g + \mathsf{T}_v$, then T will not be executed but the sender's balance $\mathsf{S}(\mathsf{T})_b$ will be reduced by $\max\left\{\sigma\left[\mathsf{S}(\mathsf{T})\right]_b, \mathsf{T}_p\mathsf{T}_g\right\}$ and nonce $\mathsf{S}(\mathsf{T})_n$ increased by one.

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 at this moment.

The transaction T can never be invalid from this point on, though it may fail in the execution.

5.3 Transaction Execution

The execution of the transaction T causes an irrevocable changed to the state σ: the nonce of the sender, $\mathsf{S}(\mathsf{T})_n$, is incremented by one and the balance $\mathsf{S}(\mathsf{T})_b$ is reduced by the gas purchasing cost $\mathsf{T}_p\mathsf{T}_g$.

We define the checkpoint state σ_0:

$$\sigma_0 \equiv \sigma \qquad \text{except:} \tag{33}$$

$$\sigma_0\left[\mathsf{S}(\mathsf{T})\right]_b \equiv \sigma\left[\mathsf{S}(\mathsf{T})\right]_b - \mathsf{T}_p\mathsf{T}_g \tag{34}$$

$$\sigma_0\left[\mathsf{S}(\mathsf{T})\right]_n \equiv \sigma\left[\mathsf{S}(\mathsf{T})\right]_n + 1 \tag{35}$$

The gas available for the proceeding computation is $g \equiv \mathsf{T}_g - g_0$, where g_0 is the intrinsic cost of T. Within this gas limit, Conflux executes the transaction depending on its type: either contract creation or message call.

5.3.1 Execution Substate

The *transaction substate* is a four tuple

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{t}}, A_r) \tag{36}$$

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.

- $A_{\mathbf{t}}$ is the set of touched accounts, of which the empty ones will be deleted on the transaction's completion.
- A_r is the refund balance. It is left for compatibility though not activated in our current version.

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

$$A^0 \equiv (\emptyset, (), \emptyset, 0) \tag{37}$$

5.3.2 Contract Creation

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

- sender s;
- original transactor o;
- available gas g;
- gas price p;
- endowment v;
- initialization EVM code $\mathbf{i}$ 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.

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 remaining gas g', the accrued substate A, the result of creation, and the output $\mathbf{o}$.

$$\left(\sigma', g', A, z, \mathbf{o}\right) \equiv \Lambda(\sigma, s, o, g, p, v, \mathbf{i}, e, w) \tag{38}$$

The address a of the newly created account α is defined as the rightmost 160 bits (i.e. the 96-th to 255-th bit) of the Keccak hash of the RLP encoding of the structure containing only the sender's address s and its account nonce.

$$a = \alpha_{addr} \equiv \mathsf{KEC}\Big(\mathsf{RLP}\big((s, \sigma[s]_n - 1)\big)\Big)[96 \ldots 255] \tag{39}$$

Note we use one fewer than the sender's nonce value since we have incremented the sender account's nonce by one prior to this call, and the value $\sigma[s]_n - 1$ is the sender's nonce at the generation of the responsible transaction or VM operation.

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 value passed (there is enough balance left as verified in section 5.2, otherwise the transaction will not be executed). Thus the mutated state becomes σ^*:

$$\sigma^* \equiv \sigma \qquad \text{except:} \tag{40}$$

$$\sigma^*[a] \equiv (1, v + \sigma[a]_b, \mathsf{TRIE}(\emptyset), \mathsf{KEC}(())) \tag{41}$$

$$\sigma^*[s] \equiv \begin{cases} \emptyset & \text{if } \sigma[s] = \emptyset \wedge v = 0 \\ (\sigma[s]_n, \sigma[s]_b - v, \sigma[s]_s, \sigma[s]_c) & \text{otherwise} \end{cases} \tag{42}$$

where $\sigma[a]_b = 0$ if $\sigma[a] = \emptyset$ when the account with address a does not exist before this message call. In case the account already exist, the pre-existing value is preserved.

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 $\mathbf{o}$.

$$(\sigma^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\sigma^*, g, I, \{s, a\}) \tag{43}$$

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

$$I_a \equiv a \tag{44}$$

$$I_o \equiv o \tag{45}$$

$$I_p \equiv p \tag{46}$$

$$I_{\mathbf{d}} \equiv () \tag{47}$$

$$I_s \equiv s \tag{48}$$

$$I_v \equiv v \tag{49}$$

$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{50}$$

$$I_e \equiv e \tag{51}$$

$$I_w \equiv w \tag{52}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_H has no special treatment and is determined from the blockchain.

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 we say an out-of-gas (OOG) exception has occurred: the evaluated state is set to the empty set $\emptyset$, 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.

If the initialization code completes successfully, a final contract-creation cost is paid for depositing the code. The code-deposit cost c is proportional to the code size of the created contract:

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{53}$$

In case there is not sufficiently remaining gas to pay this, i.e. $g^{**} < c$, then we also declare an out-of-gas exception occurs and handle it as a failed contract creation attempt.

If the contract creation fails for any reason, the value of the transaction is not transferred to the aborted contract. However, the cost of purchasing gas is not (even partly) refundable. Thus we formally specify the resultant state, gas, substate, and status code as (σ', g', A, z) as follows:

$$g' \equiv 0 \tag{54}$$

$$\sigma' \equiv \begin{cases} \sigma & \text{if } F \\ \sigma^{**} \text{ except: } \sigma'[a] = \emptyset & \text{if } \mathsf{DEAD}(\sigma^{**}, a) \\ \sigma^{**} \text{ except: } \sigma'[a]_c = \mathsf{KEC}(\mathbf{o}) & \text{otherwise} \end{cases} \tag{55}$$

$$z \equiv \begin{cases} 0 & \text{if } \sigma^{**} = \emptyset \vee g^{**} < c \\ 1 & \text{otherwise} \end{cases} \tag{56}$$

where $F \equiv ((\sigma^{**} = \emptyset \wedge \mathbf{o} = \emptyset) \vee g^{**} < c \vee |\mathbf{o}| > 24576)$ is the event that the contract creation attempt fails. In the determination of σ', the byte sequence $\mathbf{o}$, resultant from the execution of the initialization code $\mathbf{i}$, specifies the final body code for the newly created account. In case the contract creation fails eventually and hence $\sigma'[a]$ is set to $\emptyset$, Conflux *does not* revert other changes caused by executing the initialization code. z is the indicator of whether the contract creation succeeds.

Note that the intention is that the result is either a successfully created new contract with its endowment, or no new contract and no transfer of value 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 SELFDESTRUCT instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

5.3.3 Message Call

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

- sender s;
- original transactor o;
- recipient r;
- 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 $\mathbf{o}$ may be generated. In case of executing transactions the output data $\mathbf{o}$ is ignored, however message calls 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).

$$\left(\sigma', g', A, z, \mathbf{o}\right) \equiv \Theta(\sigma, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e, w) \tag{57}$$

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_1 denote the first transitional state, which is the same as the original state except for the value transferred from sender s to recipient r (if $s \neq r$):

$$\sigma_1[r]_b \equiv \sigma[r]_b + v \qquad \wedge \qquad \sigma_1[s]_b \equiv \sigma[s]_b - v \tag{58}$$

In particular, if $\sigma[r]$ was originally undefined, Conflux will treat is 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 $\sigma_1[r]$. Thus the previous equation should be taken to mean:

$$\sigma_1 \equiv \sigma_1' \qquad \text{except:} \tag{59}$$

$$\sigma_1[s] \equiv \begin{cases} \emptyset & \text{if } \sigma_1'[s] = \emptyset \wedge v = 0 \\ \alpha_1 & \text{otherwise} \end{cases} \tag{60}$$

$$\alpha_1 \equiv \left(\sigma_1'[s]_n, \sigma_1'[s]_b - v, \sigma_1'[s]_s, \sigma_1'[s]_c\right) \tag{61}$$

and $$\sigma_1' \equiv \sigma \qquad \text{except:} \tag{62}$$

$$\sigma_1'[r] \equiv \begin{cases} \left(0, v, \mathsf{TRIE}(\emptyset), \mathsf{KEC}\left(()\right)\right) & \text{if } \sigma[r] = \emptyset \wedge v \neq 0 \\ \emptyset & \text{if } \sigma[r] = \emptyset \wedge v = 0 \\ \alpha_1' & \text{otherwise} \end{cases} \tag{63}$$

$$\alpha_1' \equiv \left(\sigma_1'[r]_n, \sigma_1'[r]_b + v, \sigma_1'[r]_s, \sigma_1'[r]_c\right) \tag{64}$$

The recipient's associated code $\mathbf{b}$, whose Keccak hash is $\sigma[r]_c$, is executed according to the execution model. Note that the pair $(\mathsf{KEC}(\mathbf{b}), \mathbf{b})$ must be stored at some previous point, i.e. at the last update of the code hash $\sigma[r]_c$ of the recipient's account. Thus $\mathbf{b}$ can be efficiently determined from $\sigma[r]_c$, and it is unique following the collision resistance of KEC.

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:

$$\sigma' \equiv \begin{cases} \sigma & \text{if } \sigma^{**} = \emptyset \\ \sigma^{**} & \text{otherwise} \end{cases} \tag{65}$$

$$g' \equiv \begin{cases} 0 & \text{if } \sigma^{**} = \emptyset \wedge \mathbf{o} = \emptyset \\ g^{**} & \text{otherwise} \end{cases} \tag{66}$$

$$z \equiv \begin{cases} 0 & \text{if } \sigma^{**} = \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{67}$$

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_1.

$$(\sigma^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi\left(\sigma_1, g, I, \{s, t\}\right) \tag{68}$$

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

$$I_a \equiv r \tag{69}$$

$$I_o \equiv o \tag{70}$$

$$I_p \equiv p \tag{71}$$

$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{72}$$

$$I_s \equiv s \tag{73}$$

$$I_v \equiv \tilde{v} \tag{74}$$

$$I_{\mathbf{b}} \equiv \mathbf{b} \tag{75}$$

$$I_e \equiv e \tag{76}$$

$$I_w \equiv w \tag{77}$$

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 contracts" with reserved recipient's address $r \in \{1, 2, \ldots, 8\}$. In the present work these exceptional contracts are specified as in the latest version of Ethereum [3].

5.4 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 is intrinsically bounded by the limited amount of available gas. For this moment we implement the well-known Ethereum Virtual Machine (EVM), and the execution model follows [3].

5.4.1 Basics

The EVM is a stack-based architecture with 256-bit word size. The stack has a maximum size of 1024. 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.

5.4.2 Gas Consumption

The cost of execution, aka. *gas*, is charged under three distinct 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;
3. the increase in the memory usage.

In particular, the total gas for memory-usage payable is proportional to smallest multiple of 32 bytes that are required to include all memory indices (whether for read or write). This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional cost.

We remark that (at least in the current version) the execution fee for an operation that clears a storage entry is charged normally, i.e. there is no refund. This is distinct from the gas refunding policy in Ethereum [3].

5.4.3 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 transactor who originated this execution.
- 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, this would be the transaction sender's address.
- 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.

The state transition is defined by the execution function Ξ, which takes as input the current 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:

$$\left(\sigma', g', A, \mathbf{o}\right) \equiv \Xi\left(\sigma, g, I\right) \tag{78}$$

where we recall that the accrued state A consists of the selfdestructs set $\mathbf{s}$, the log series $\mathbf{l}$ and the touched accounts $\mathbf{t}$:

$$A \equiv (\mathbf{s}, \mathbf{l}, \mathbf{t}) \tag{79}$$

Note that in the current version there is no refund for destructing contracts or accounts. This may be changed in a later version.

5.4.4 Execution Overview

We omit the detailed definition of the execution function Ξ here, since it follows exactly the same definition as in Ethereum yellowpaper [3].

6. Proof of Work

In this version, we tentatively set the proof of work function PoW as SHA2-256. It will be updated before the launch of Conflux mainnet.

6.1 Difficulty

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 200 epochs and the average timestamps of blocks in the beginning and ending epochs, and then set the target difficulty for the next 200 epochs such that the expected block generation rate should be roughly one block per 5 seconds.

Formally, for $0 \leq j \leq 200$, the target difficulty of a block at height j is set to $\mathbf{d}_0 \equiv 5 \times 10^9 = 5000000000$; for any positive integer $i \geq 1$, the target difficulty of blocks at height $j \in [200i+1, 200i+200]$ is set to

$$\mathbf{d}_i \equiv \mathbf{d}_{i-1} \times 5 \times \sum_{j=1}^{200} |\textsc{Epoch}_{200i+j}| / \left(\frac{\sum_{\mathsf{B}' \in \textsc{Epoch}_{200i}} \mathsf{B}'_{\mathsf{H}_s}}{|\textsc{Epoch}_{200i}|} - \frac{\sum_{\mathsf{B}' \in \textsc{Epoch}_{200(i-1)}} \mathsf{B}'_{\mathsf{H}_s}}{|\textsc{Epoch}_{200(i-1)}|} \right) \tag{80}$$

where for every $k \in \mathbb{N}$, $\textsc{Epoch}_k$ denotes the set of blocks included in the k-th epoch and $|\textsc{Epoch}_k|$ is the number of blocks in that epoch.

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 own view of blocks in $\mathsf{PAST}(\mathsf{B})$. In particular, a block B at height $h \equiv \mathsf{B}_{\mathsf{H}_h}$ should have target difficulty

$$\mathsf{B}_{\mathsf{H}_d} \equiv \begin{cases} \mathbf{d}_0 & h = 0 \\ \mathbf{d}_{\lfloor \frac{h-1}{200} \rfloor} & \text{o.w.} \end{cases} \tag{81}$$

where $\mathbf{d}_i$'s are calculated with respect to $\mathsf{PAST}(\mathsf{B})$.

However, as soon as all nodes agree on the pivot block 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 B_k. Formally,

$$\mathbf{d}_{\textsc{Epoch}_k} \equiv \mathsf{H}_d(\mathsf{B}_k) \tag{82}$$

where $\mathsf{H}_d(\mathsf{B}_k)$ is the **difficulty** field in B_k's header and it equals to $\mathbf{d}_k$ derived from the past view of B_k.

7. 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.

7.1 Base Block Award

The amount of coins issued to miners in every block is set to a value in accordance to the mining schedule as stated in Section **??**. In what follows we refer to this value as the *base block award* or simply *base award*, and denote it by R_{base}. The base award is a global parameter that decreases over time until the inflation rate drops below 2 percent. More specifically, it decreases every 1572480 blocks, which is equivalent to 91 days in expectation. Note that the decrease of base award may happen in the middle of an epoch.

7.1.1 Difficulty Discount

If a block B in an epoch $\textsc{Epoch}_k$ satisfies the epoch's target difficulty, i.e. $\mathsf{B}_d \geq \mathbf{d}_{\textsc{Epoch}_k}$, then block B is entitled the base award R_{base} of that epoch, denoted by $\mathsf{R}_{base}(\mathsf{B}) \equiv \mathsf{R}_{base}(\textsc{Epoch}_k)$.

If a block B in $\textsc{Epoch}_k$ has a lower target difficulty, i.e. $\mathsf{B}_d < \mathbf{d}_{\textsc{Epoch}_k}$, then we decide the base award of B by its *actual difficulty*: it gets normal base award if its actual difficulty reaches the epoch's target difficulty $\mathbf{d}_{\textsc{Epoch}_k}$, and zero base award $\mathsf{R}_{base}(\mathsf{B}) \equiv 0$ in case the actual difficulty is less than $\mathbf{d}_{\textsc{Epoch}_k}$. Note that the expected base award is the same as when setting $\mathsf{R}_{base}(\mathsf{B}) \equiv \frac{\mathsf{B}_d}{\mathbf{d}_{\textsc{Epoch}_k}} \cdot \mathsf{R}_{base}(\textsc{Epoch}_k)$.

For convenience, we introduce $\mathsf{BF}(\mathsf{B})$ to denote the *base factor* of the multiple of base award that B is entitled.

$$\mathsf{BF}(\mathsf{B}) \equiv \begin{cases} 1 & \mathsf{B} \text{ is valid and its actual difficulty is greater or equal than } \mathbf{d}_{\text{EPOCH}_k} \\ 0 & \text{otherwise (either } \mathsf{B} \text{ has actual difficulty less than } \mathbf{d}_{\text{EPOCH}_k} \text{ or } \mathsf{B} \text{ is only partially valid)} \end{cases} \tag{83}$$

As a consequence, $\mathsf{R}_{base}(\mathsf{B}) = \mathsf{BF}(\mathsf{B}) \cdot \mathsf{R}_{base}(\text{EPOCH}(\mathsf{B})) = \mathsf{BF}(\mathsf{B}) \cdot \mathsf{R}_{base}$.

7.1.2 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. Let $\mathscr{A}(\mathsf{B})$ denote the set of anti-cone blocks of B within 10 epochs after the epoch of B. Then the *anti-cone penalty factor of* B is defined as

$$\mathsf{AF}(\mathsf{B}) \equiv 1 - \frac{\left(W(\mathscr{A}(\mathsf{B}))/\mathbf{d}_{\text{EPOCH}(\mathsf{B})}\right)^2}{\gamma} \tag{84}$$

where $\gamma \equiv 10000$ is a fixed constant and $W(\mathscr{A}(\mathsf{B}))$ refers to the total difficulty of blocks in $\mathscr{A}(\mathsf{B})$. We remark that $W(\mathscr{A}(\mathsf{B}))/\mathbf{d}_{\text{EPOCH}(\mathsf{B})}$ is the equivalent number of blocks in the anti-cone of B, which corresponds to the potion of computing power in B's anti-cone.

This anti-cone penalty factor is introduced to incentivize inclusion of ommer 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 ommer blocks, nor discount in block award for those non-pivot blocks.

7.2 Transaction Fee Reward

If a transaction T is first included in the i-th epoch 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: a) $\forall \mathsf{B}' \in \mathsf{PAST}(\mathsf{B}), \mathsf{T} \notin \mathsf{B}'_{\mathsf{Ts}}$; and b) B belongs to EPOCH_i (the epoch that T is first included in).

The transaction fee is distributed proportionally to those blocks with respect to their weight defined by actual difficulty (compared to the epoch's target difficulty) as defined in section 7.1.1. That is, a block with a fully valid header and actual difficulty above the epoch target difficulty gets weight 1 , and other blocks with partially valid headers or lower-than-target actual difficulty gets weight 0.

In particular, if the transaction T is only included in blocks of 0 weight, the transaction fee is burnt although the transaction will still be processed.

The total transaction fee reward of a block B is the sum of its portion of fees from every transaction in B_{Ts}.

$$\mathsf{R}_{fee}(\mathsf{B}) \equiv \sum_{\mathsf{T}\,:\,\mathsf{B} \text{ properly includes } \mathsf{T}} \mathsf{T}_p \mathsf{T}_g \cdot \frac{\mathsf{BF}(\mathsf{B})}{\sum_{\mathsf{B}':\mathsf{B}' \text{ properly includes } \mathsf{T}} \mathsf{BF}(\mathsf{B}')} \tag{85}$$

7.3 Final Reward to Miners

The final mining reward of a block B will be added to the beneficiary's account as specified in B_c. The reward effects at the end of 7 epochs after the epoch of B. Due to 5-blocks deferred execution, this reward becomes avaliable at $7+5$ epochs after the epoch of B. For example, if B appears in EPOCH_i, then the account B_c receives the mining reward at the end of EPOCH_{i+7}. Such a receive will be executed at the end of EPOCH_{i+12}. The total mining reward is calculated as follows:

$$\mathsf{R}(\mathsf{B}) \equiv \mathsf{AF}(\mathsf{B}) \cdot \mathsf{R}_{base}(\mathsf{B}) + \mathsf{R}_{fee}(\mathsf{B}) \tag{86}$$

In particular, note that $\mathsf{R}(\mathsf{B}) = 0$ as long as $\mathsf{BF}(\mathsf{B}) = 0$.

8. Concrete Protocol Implementation

Concretely, we set the following parameters for Conflux. Note that the numbers here are only for the most recent testnet and may change in the future.

Parameter	Value
Block time	5 s
Suggested block size bound	4 MB
Starting coinbase award	900 Conflux
Starting difficulty ($\mathbf{d}_0$)	$5 \times 10^9 = 5000000000$
Anti-cone penalty factor (γ)	10000
Deferred execution	5 epochs
Mining reward freezing time	7 epochs

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] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870*, 2018.