Polkadot Runtime Environment

Protocol Specification

May 10, 2019

1 Conventions and Definitions

Definition 1. ***Runtime*** *is the state transition function of the decentralized ledger protocol.*

Definition 2. *A* ***path graph*** *or a* ***path*** *of n nodes formally referred to as $\boldsymbol{P_n}$, is a tree with two nodes of vertex degree 1 and the other n-2 nodes of vertex degree 2. Therefore, P_n can be represented by sequences of $(v_1, ..., v_n)$ where $e_i = (v_i, v_{i+1})$ for $1 \leqslant i \leqslant n - 1$ is the edge which connect v_i and v_{i+1}.*

Definition 3. ***Radix-r tree*** *is a variant of a trie in which:*

- *Every node has at most r children where $r = 2^x$ for some x;*
- *Each node that is the only child of a parent, which does not represent a valid key is merged with its parent.*

As a result, in a radix tree, any path whose interior vertices all have only one child and does not represent a valid key in the data set, is compressed into a single edge. This improves space efficiency when the key space is sparse.

Definition 4. *By a* ***sequences of bytes*** *or a* ***byte array****, b, of length n, we refer to*

$$b := (b_0, b_1, ..., b_{n-1}) \text{ such that } 0 \leqslant b_i \leqslant 255$$

We define $\mathbb{B}_n$ to be the ***set of all byte arrays of length n****. Furthermore, we define:*

$$\mathbb{B} := \bigcup_{i=0}^{\infty} \mathbb{B}_i$$

Notation 5. *We represent the concatenation of byte arrays $a := (a_0, ..., a_n)$ and $b := (b_0, ..., b_m)$ by:*

$$a || b := (a_0, ..., a_n, b_0, ..., b_m)$$

Definition 6. *For a given byte b the* ***bitwise representation*** *of b is defined as*

$$b := b^7 \ldots b^0$$

where

$$b = 2^0 b^0 + 2^1 b^1 + \cdots + 2^7 b^7$$

Definition 7. *By the* ***little-endian*** *representation of a non-negative integer, I, represented as*

$$I = (B_n \ldots B_0)_{256}$$

in base 256, we refer to a byte array $B = (b_0, b_1, \ldots, b_n)$ such that

$$b_i := B_i$$

Accordingly, define the function $\mathrm{Enc}_{\mathrm{LE}}$*:*

$$\begin{array}{rlcl}\mathrm{Enc}_{\mathrm{LE}}: & \mathbb{Z}^+ & \rightarrow & \mathbb{B} \\ & (B_n \ldots B_0)_{256} & \mapsto & (B_{0,} B_1, \ldots, B_n)\end{array}$$

Definition 8. *By* **`UINT32`** *we refer to a non-negative integer stored in a byte array of length 4 using little-endian encoding format.*

Definition 9. *A* ***blockchain*** *C is a directed path graph. Each node of the graph is called* ***Block*** *and indicated by* $\boldsymbol{B}$*. The unique sink of C is called* ***Genesis Block****, and the source is called the* ***Head*** *of C. For any vertex (B_1, B_2) where $B_1 \rightarrow B_2$ we say B_2 is the* ***parent*** *of B_1 and we indicate it by*

$$B_2 := P(B_1)$$

1.1 Block Tree

In the course of formation of a (distributed) blockchain, it is possible that the chain forks into multiple subchains in various block positions. We refer to this structure as a *block tree:*

Definition 10. *The* ***block tree*** *of a blockchain, denoted by* BT *is the union of all different versions of the blockchain observed by all the nodes in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2.*

Definition 11 gives the means to highlight various branches of the block tree.

Definition 11. *Let G be the root of the block tree and B be a node of it. By* **Chain($\boldsymbol{B}$)**, *we refer to the path graph from G to B in* BT*. If B' is another node on* Chain(B)*, then by* SubChain(B', B) *we refer to the subgraph of* Chain(B) *path graph which contains both B and B'.* Longest-Path(BT) *returns a path graph of* BT *which is the longest among all paths in* BT*.* Deepest-Leaf(BT) *returns the head of* Longest-Path(BT) *chain.*

Because every block in the blockchain contains a reference to its parent, it is easy to see that the block tree is de facto a tree.

A block tree naturally imposes partial order relationships on the blocks as follows:

Definition 12. *We say* ***B is descendant of B'****, formally noted as* $\boldsymbol{B > B'}$ *if B is a descendant of B' in the block tree.*

2 Block Format

In Polkadot RE, a block is made of two main parts, namely the *block header* and the *list of extrinsics*. *The Extrinsics* represent the generalization of the concept of *transaction*, containing any set of data that is external to the system, and which the underlying chain wishes to validate and keep track of.

2.1 Block Header

The block header is designed to be minimalistic in order to boost the efficiency of the light clients. It is defined formally as follows:

Definition 13. *The **header of block B**,* **Head(*B*)** *is a 5-tuple containing the following elements:*

- ***parent_hash:*** *is the 32-byte Blake2b hash (see Section 8.2) of the header of the parent of the block indicated henceforth by* $\boldsymbol{H_p}$.
- ***number:*** *formally indicated as* $\boldsymbol{H_i}$ *is an integer, which represents the index of the current block in the chain. It is equal to the number of the ancestor blocks. The genesis block has number 0.*
- ***state_root:*** *formally indicated as* $\boldsymbol{H_r}$ *is the root of the Merkle trie, whose leaves implement the storage for the system.*
- ***extrinsics_root:*** *is the field which is reserved for the runtime to validate the integrity of the extrinsics composing the block body. For example, it can hold the root hash of the Merkle trie which stores an ordered list of the extrinsics being validated in this block. The* extrinsics_root *is set by the runtime and its value is opaque to Polkadot RE. This element is formally referred to as* $\boldsymbol{H_e}$.
- ***digest:*** *this field is used to store any chain-specific auxiliary data, which could help the light clients interact with the block without the need of accessing the full storage. Polkadot RE does not impose any limitation or specification for this field. Essentially, it can be a byte array of any length. This field is indicated as* $\boldsymbol{H_d}$

Definition 14. *The **Block Header Hash of Block B**,* $\boldsymbol{H_h(b)}$*, is the hash of the header of block B encoded by simple codec:*

$$H_b(b) := \text{Blake2}b(\text{Enc}_{\text{SC}}(\text{Head}(B)))$$

2.2 Justified Block Header

The Justified Block Header is provided by the consensus engine and presented to the Polkadot RE, for the block to be appended to the blockchain. It contains the following parts:

- **block_header** the complete block header as defined in Section 2.1 and denoted by Head(B).
- **justification**: as defined by the consensus specification indicated by Just(B) [link this to its definition from consensus].
- **authority Ids**: This is the list of the Ids of authorities, which have voted for the block to be stored and is formally referred to as $A(B)$. An authority Id is 32bit.

2.3 Processing Extrinsics

The block body consists of a set of extrinsics. Nonetheless, Polkadot RE does not specify or limit the internals of each extrinsics. From Polkadot RE point of view, each extrinsics is a SCALE encoded in byte arrays (see Definition 53).

The extrinsics are submitted to the node through the *transactions* network message specified in Section 4.4.1. Upon receiving a transactions message, Polkadot RE separates the submitted transactions message into individual extrinsics and runs Algorithm 1 to validate them and store them to include them into future blocks.

Algorithm 1. VALIDATE-EXTRINSICS-AND-STORE(L: list of extrinsics)

```
for $E$ in $L$
    $B_d \leftarrow$ DEEPEST-LEAF(BT)
    $N \leftarrow H_n(B_d)$
    $R \leftarrow$ CALL-RUNTIME-ENTRY(TaggedTransactionQueue_validate_transaction, $N, E$)
    if $R$ indicates $E$ is Valid
        ADD-TO-EXTRINSIC-QUEUE($E, R$)
```

2.4 Extrinsic Queue

[To be specced]

2.5 Block Format

3 Interactions with the Runtime

Runtime is the code implementing the logic of the chain. This code is decoupled from the Polkadot RE to make the Runtime easily upgradable without the need to upgrade the Polkadot RE itself. In this section, we describe the details upon which the Polkadot RE is interacting with the Runtime.

3.1 Loading the Runtime code

Polkadot RE expects to receive the code for the runtime of the chain as a compiled WebAssembly (Wasm) Blob. The current runtime is stored in the state database under the key represented as a byte array:

$$b := \text{3A,63,6F,64,65}$$

which is the byte array of ASCII representation of string ":code" (see Section 11). For any call to the runtime, Polkadot RE makes sure that it has the most updated Runtime as calls to runtime have potentially the ability to change the runtime code.

The initial runtime code of the chain is embedded as an extrinsics into the chain initialization JSON file and is submitted to Polkadot RE (see Section 10).

Subsequent calls to the runtime have the ability to call the storage API (see Section A) to insert a new Wasm blob into runtime storage slot to upgrade the runtime.

3.2 Code Executor

Polkadot RE provides a Wasm Virtual Machine (VM) to run the Runtime. The Wasm VM exposes the Polkadot RE API to the Runtime, which, on its turn, executes a call to the Runtime entries stored in the Wasm module. This part of the Runtime environment is referred to as the ***Executor***.

Definition 15 introduces the notation for calling the runtime entry which is used whenever an algorithm of Polkadot RE needs to access the runtime.

Notation 15. *By*

$$\text{Call-Runtime-Entry}(\texttt{Runtime-Entry}, A_1, A_2, \ldots, A_n)$$

we refer to the task using the execuping me cherie when the edits are readytor to invoke the `Runtime-Entry` *while passing an $A_1, \ldots, A_n$ argument to it and using the encoding described in Section 3.2.2.*

In this section, we specify the general setup for an Executor call into the Runtime. In Section 3.3 we specify the parameters and the return values of each Runtime entry separately.

3.2.1 Access to Runtime API

When Polkadot RE calls a Runtime entry it should make sure Runtime has access to the all Polkadot Runtime API functions described in Appendix A. This can be done for example by loading another Wasm module alongside the runtime which imports these functions from Polkadot RE as host functions.

3.2.2 Sending Arguments to Runtime

In general, all data exchanged between Polkadot RE and the Runtime is encoded using SCALE codec described in Section 9.1. As a Wasm function, all runtime entries have the following identical signatures:

```
(func $runtime_entry (param $data i32) (param $len i32) (result i64))
```

In each invocation of a Runtime entry, the arguments which are supposed to be sent to the entry, need to be encoded using SCALE codec into a byte array B using the procedure defined in Definition 9.1.

The Executor then needs to retrieve the Wam memory buffer of the Runtime Wasm module and extend it to fit the size of the byte array. Afterwards, it needs to copy the byte array B value in the correct offset of the extended buffer. Finally, when the Wasm method `runtime_entry`, corresponding to the entry is invoked, two UINT32 integers are sent to the method as arguments. The first argument `data` is set to the offset where the byte array B is stored in the Wasm the extended shared memory buffer. The second argument `len` sets the length of the data stored in B., and the second one is the size of B.

3.2.3 The Return Value from a Runtime Entry

The value which is returned from the invocation is an `i64` integer, representing two consecutive `i32` integers in which the least significant one indicates the pointer to the offset of the result returned by the entry encoded in SCALE codec in the memory buffer. The most significant one provides the size of the blob.

In the case that the runtime entry is returning a boolean value, then the SCALEd value returns in the least significant byte and all other bytes are set to zero.

3.3 Entries into Runtime

Polkadot RE assumes that at least the following functions are implemented in the Runtime Wasm blob and has been exported as shown in Snippet 1:

```
(export "Core_version" (func $Core_version))
(export "Core_authorities" (func $Core_authorities))
(export "Core_execute_block" (func $Core_execute_block))
(export "Core_initialise_block" (func $Core_initialise_block))
(export "Metadata_metadata" (func $Metadata_metadata))
(export "BlockBuilder_apply_extrinsic" (func $BlockBuilder_apply_extrinsic))
(export "BlockBuilder_finalise_block" (func $BlockBuilder_finalise_block))
(export "BlockBuilder_inherent_extrinsics"
        (func $BlockBuilder_inherent_extrinsics))
(export "BlockBuilder_check_inherents" (func $BlockBuilder_check_inherents))
(export "BlockBuilder_random_seed" (func $BlockBuilder_random_seed))
(export "TaggedTransactionQueue_validate_transaction"
        (func $TaggedTransactionQueue_validate_transaction))
(export "OffchainWorkerApi_offchain_worker"
        (func $OffchainWorkerApi_offchain_worker))
(export "ParachainHost_duty_roster" (func $ParachainHost_duty_roster))
(export "ParachainHost_active_parachains"
        (func $ParachainHost_active_parachains))
(export "ParachainHost_parachain_head" (func $ParachainHost_parachain_head))
(export "ParachainHost_parachain_code" (func $ParachainHost_parachain_code))
(export "GrandpaApi_grandpa_pending_change"
        (func $GrandpaApi_grandpa_pending_change))
(export "GrandpaApi_grandpa_forced_change"
        (func $GrandpaApi_grandpa_forced_change))
(export "GrandpaApi_grandpa_authorities"
        (func $GrandpaApi_grandpa_authorities))
(export "ParachainHost_validators" (func $Core_authorities))
(export "BabeApi_slot_duration" (func $BabeApi_slot_duration))
(export "BabeApi_slot_winning_threshold"
        (func $BabeApi_slot_winning_threshold))
```

Snippet 1. Snippet to export entries into tho Wasm runtime module

The following sections describe the standard based on which Polkadot RE communicates with each runtime entry.

3.3.1 `Core_version`

This entry receives no argument; it returns the version data encoded in ABI format described in Section 3.2.3 containing the following data:

Name	Type	Description
`spec_name`	String	runtime identifier
`impl_name`	String	the name of the implementation (e.g. C++)
`authoring_version`	UINT32	the version of the authorship interface
`spec_version`	UINT32	the version of the runtime specification
`impl_version`	UINT32	the version of the runtime implementation
`apis`	ApisVec	List of supported AP

Table 1. Detail of the version data type returns from runtime `version` function

3.3.2 `Core_authorities`

This entry is to report the set of authorities at a given block. It receives `block_id` as an argument; it returns an array of `authority_id`'s.

3.3.3 `Core_execute_block`

This entry is responsible for executing all extrinsics in the block and reporting back the changes into the state storage. It receives the block header and the block body as its arguments, and it returns a triplet:

Name	Type	Description
`results`	Boolean	Indicating if the execution was su
`storage_changes`	[???]	Contains all changes to the state storage
`change_updat`	[???]	

Table 2. Detail of the data execute_block returns after execution

3.3.4 `Core_initialise_block`

3.3.5 `TaggedTransactionQueue_validate_transaction`

[Explain function]

4 Network Interactions

4.1 Extrinsics Submission

Extrinsic submission is made by sending a *Transactions* network message. The structure of this message is specified in Section 4.4.1.

Upon receiving an Transactions message, Polkadot RE decodes the transaction and calls `validate_trasaction` runtime function defined in Section 3.3.5, to check the validity of the extrinsic. If `validate_transaction` considers the submitted extrinsics as a valid one, Polkadot RE makes the extrinsics available for the consensus engine for inclusion in future blocks.

4.2 Network Messages

This section specifies various types of messages which Polkadot RE receives from the network. Furthermore, it also explains the appropriate responses to those messages.

4.3 General structure of network messages

Definition 16. *A* ***network message*** *is a byte array,* $\boldsymbol{M}$ *of length* $\|M\|$ *such that:*

$$\begin{array}{cc} M_1 & \text{Message Type Indicator} \\ M_2 \ldots M_{\|M\|} & \text{Enc}_{\text{SC}}(\text{MessageBody}) \end{array}$$

The body of each message consists of different components based on its type. The different possible message types are listed below in Table 3. We describe the sub-components of each message type individually in Section 4.4.

M_1	Message Type	Description
0	Status	
1	Block Request	
2	Block Response	
3	Block Announce	
4	Transactions	
5	Consensus	
6	Remote Call Request	
7	Remote Call Response	
8	Remote Read Request	
9	Remote Read Response	
10	Remote Header Request	
11	Remote Header Response	
12	Remote Changes Request	
13	Remote Changes Response	
255	Chain Specific	

Table 3. List of possible network message types

4.4 Detailed Message Structure

This section disucsses the detailed structure of each network message.

4.4.1 Transactions

The transactions Message is represented by M_T and is defined as follows:

$$M_T := \text{Enc}_{\text{SC}}(C_1, \ldots, C_n)$$

in which:

$$C_i := \text{Enc}_{\text{SC}}(E_i)$$

Where each E_i is a byte array and represents a sepearate extrinsic. Polkadot RE is indifferent about the content of an extrinsic and treats is as a blob of data.

4.5 Block Submission and Validation

Block validation is the process, by which the client asserts that a block is fit to be added to the blockchain. This means that the block is consistent with the world state and transitions from the state of the system to a new valid state.

Blocks can be handed to the Polkadot RE both from the network stack and from the consensus engine.

Both the Runtime and the Polkadot RE need to work together to assure block validity. This can be accomplished by Polkadot RE invoking `execute_block` entry into the runtime as a part of the validation process.

Polkadot RE implements the following procedure to assure the validity of the block:

Algorithm 2. Import-and-Validate-Block($B, \text{Just}(B)$)

```
Verify-Block-Justification($B, \text{Just}(B)$)
if $B$ is Finalized and $P(B)$ is not Finalized
    Mark-as-Final($P(B)$)
Verify $H_{p(B)} \in$ Blockchain
State-Changes = Runtime($B$)
Update-World-State(State-Changes)
```

For the definition of the finality and the finalized block see Section 7.2.

5 State Storage and the Storage Trie

For storing the state of the system, Polkadot RE implements a hash table storage where the keys are used to access each data entry. There is no assumption either on the size of the key nor on the size of the data stored under them, besides the fact that they are byte arrays with specific upper limits on their length. The limit is imposed by the encoding algorithms to store the key and the value in the storage trie.

5.1 Accessing The System Storage

Polkadot RE implements various functions to facilitate access to the system storage for the runtime. Section A lists all of those functions. Here we formalize the access to the storage when it is being directly accessed by Polkadot RE (in contrast to Polkadot runtime).

Definition 17. *The* ***StoredValue*** *function retrieves the value stored under a specific key in the state storage and is formally defined as :*

$$\begin{array}{rc}\text{StoredValue:} & \mathcal{K} \rightarrow \mathcal{V} \\ & k \mapsto \begin{cases} v & \textit{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases}\end{array}$$

where $\mathcal{K} \subset \mathbb{B}$ and $\mathcal{V} \subset \mathbb{B}$ are respectively the set of all keys and values stored in the state storage.

5.2 The General Tree Structure

In order to ensure the integrity of the state of the system, the stored data needs to be re-arranged and hashed in a *modified Merkle Patricia Tree*, which hereafter we refer to as the ***Trie***. This rearrangment is necessary to be able to compute the Merkle hash of the whole or part of the state storage, consistently and efficiently at any given time.

The Trie is used to compute the *state root*, H_r, (see Definition 13), whose purpose is to authenticate the validity of the state database. Thus, Polkadot RE follows a rigorous encoding algorithm to compute the values stored in the trie nodes to ensure that the computed Merkle hash, H_r, matches across the Polkadot RE implementations.

The Trie is a *radix-16* tree as defined in Definition 3. Each key value identifies a unique node in the tree. However, a node in a tree might or might not be associated with a key in the storage.

When traversing the Trie to a specific node, its key can be reconstructed by concatenating the subsequences of the key which are stored either explicitly in the nodes on the path or implicitly in their position as a child of their parent.

To identify the node corresponding to a key value, k, first we need to encode k in a consistent with the Trie structure way. Because each node in the trie has at most 16 children, we represent the key as a sequence of 4-bit nibbles:

Definition 18. *For the purpose of labeling the branches of the Trie, the key k is encoded to k_{enc} using KeyEncode functions:*

$$k_{\text{enc}} := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) := \text{KeyEncode}(k) \tag{1}$$

such that:

$$\text{KeyEncode}(k) : \begin{cases} \mathbb{B} & \rightarrow \quad \text{Nibbles}^4 \\ k := (b_1, \ldots, b_n) := & \mapsto \quad (b_1^1, b_1^2, b_2^1, b_2^2, \ldots, b_n^1, b_n^2) \\ & \quad := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) \end{cases}$$

where Nibble^4 *is the set of all nibbles of 4-bit arrays and b_i^1 and b_i^2 are 4-bit nibbles, which are the big endian representations of b_i:*

$$(b_i^1, b_i^2) := (b_i / 16, b_i \bmod 16)$$

, where mod is the remainder and / is the integer division operators.

By looking at k_{enc} as a sequence of nibbles, one can walk the radix tree to reach the node identifying the storage value of k.

5.3 The Trie structure

In this subsection, we specify the structure of the nodes in the Trie as well as the Trie structure:

Notation 19. *We refer to the* ***set of the nodes of Polkadot state trie*** *by $\mathcal{N}$. By $N \in \mathcal{N}$ to refer to an individual node in the trie.*

Definition 20. *The State Trie is a radix-16 tree. Each Node in the Trie is identified with a unique key k_N such that:*

- *k_N is the shared prefix of the key of all the descendants of N in the Trie.*

and, at least one of the following statements holds:

- *(k_N, v) corresponds to an existing entry in the State Storage.*
- *N has more than one child.*

Conversely, if (k, v) is an entry in the State Trie then there is a node $N \in \mathcal{N}$ such that k_N=k.

Notation 21. *A **branch** node is a node which has one child or more. A branch node can have at most 16 children. A **leaf** node is a childless node. Accordingly:*

$$\mathcal{N}_b := \{N \in \mathcal{N} | N \textit{ is a branch node}\}$$
$$\mathcal{N}_l := \{N \in \mathcal{N} | N \textit{ is a leaf node}\}$$

For each Node, part of k_N is built while the trie is traversed from root to N part of k_N is stored in N as formalized in Definition 22.

Definition 22. *For any $N \in \mathcal{N}$, its key k_N is divided into an **aggregated prefix key**, $\mathbf{pk_N^{Agr}}$, aggregated by Algorithm 3 and a **partial key**, $\mathbf{pk_N}$ of length $0 \leqslant l_{\mathrm{pk}_N} \leqslant 65535$ such that:*

$$\mathrm{pk}_N := (k_{\mathrm{enc}_i}, \ldots, k_{\mathrm{enc}_{i+l_{\mathrm{pk}_N}}})$$

where pk_N is a suffix subsequence of k_N; and we have:

$$\mathrm{KeyEncode}(k_N) = \mathrm{pk}_N^{\mathrm{Agr}} | \mathrm{pk}_N = (k_{\mathrm{enc}_1}, \ldots, k_{\mathrm{enc}_{i-1}}, k_{\mathrm{enc}_i}, k_{\mathrm{enc}_{i+l_{\mathrm{pk}_N}}})$$

Part of $\mathrm{pk}_N^{\mathrm{Agr}}$ is explicitly stored in N's ancestors. Additionally, for each ancestor, a single nibble is implicitly derived while traversing from the ancestor to its child included in the traversal path using the Index_N function defined in Definition 23.

Definition 23. *For $N \in \mathcal{N}_b$ and N_c child of N, we define $\mathbf{Index_N}$ function as:*

$$\begin{aligned} \mathrm{Index}_N\colon\ \{N_c \in \mathcal{N} | N_c \textit{ is a child of } N\} &\to \mathrm{Nibbles}_1^4 \\ N_c &\mapsto i \end{aligned}$$

such that

$$k_{N_c} = k_N || i || \mathrm{pk}_{N_c}$$

Assuming that P_N is the path (see Definition 2) from the Trie root to node N, Algorithm 3 rigorously demonstrates how to build $\mathrm{pk}_N^{\mathrm{Agr}}$ while traversing P_N.

Algorithm 3. Aggregate-Key(P_N: $=$(TrieRoot $= N_1, \ldots, N_j = N$))

```
$\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \phi$
$i \leftarrow 1$
while ($N_i \neq N$)
    $\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \mathrm{pk}_N^{\mathrm{Agr}} || \mathrm{pk}_N$
    $\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \mathrm{pk}_N^{\mathrm{Agr}} || \mathrm{Index}_{N_i}(N_{i+1})$
return $\mathrm{pk}_N^{\mathrm{Agr}}$
```

Definition 24. *A node $N \in \mathcal{N}$ stores the **node value**, $\boldsymbol{v_N}$, which consists of the following concatenated data:*

Node Header	Partial key	Node Subvalue

Formally noted as:

$$v_N := \mathrm{Head}_N || \mathrm{Enc}_{\mathrm{HE}}(\mathrm{pk}_N) || \mathrm{sv}_N$$

where Head_N, pk_N, $\mathrm{Enc}_{\mathrm{nibbles}}$ *and* sv_N *are defined in Definitions 25,22, 59 and 27, respectively.*

Definition 25. *The **node header** of node* N, Head_N, *consists of* $l \geqslant 1$ *bytes*

Node Type	pk length	pk length extra byte 1	pk key length extra byte 2	…	pk length extra byte l
$\mathrm{Head}_{N,1}^{6-7}$	$\mathrm{Head}_{N,1}^{0-5}$	$\mathrm{Head}_{N,2}$	….		$\mathrm{Head}_{N,l+1}$

In which $\mathrm{Head}_{N,1}^{6-7}$, *the two most significant bits of the first byte of* Head_N *are determined as follows:*

$$\mathrm{Head}_{N,1}^{6-7} := \begin{cases} 00 & \text{Special case} \\ 01 & \text{Leaf Node} \\ 10 & \text{Branch Node with } k_N \notin \mathcal{K} \\ 11 & \text{Branch Node with } k_N \in \mathcal{K} \end{cases}$$

where $\mathcal{K}$ *is defined in Definition 17.*

$\mathrm{Head}_{N,1}^{0-5}$, *the 6 least significant bits of the first byte of* Head_N *are defined to be:*

$$\mathrm{Head}_{N,1}^{0-5} := \begin{cases} \|\mathrm{pk}_N\|_{\mathrm{nib}} & \|\mathrm{pk}_N\|_{\mathrm{nib}} < 63 \\ 63 & \|\mathrm{pk}_N\|_{\mathrm{nib}} \geqslant 63 \end{cases}$$

In which $\|\mathbf{pk}_{\boldsymbol{N}}\|_{\mathbf{nib}}$ *is the length of* pk_N *in number nibbles.* $\mathrm{Head}_{N,2}, \ldots, \mathrm{Head}_{N,l}$ *bytes are determined by Algorithm 4.*

Algorithm 4. Partial-Key-Length-Encoding($\mathrm{Head}_{N,1}^{6-7}$, pk_N)

```
if $\|\mathrm{pk}_N\|_{\mathrm{nib}} \geqslant 2^{16}$
    return Error
$\mathrm{Head}_{N,1} \leftarrow 64 \times \mathrm{Head}_{N,1}^{6-7}$
if $\|\mathrm{pk}_N\|_{\mathrm{nib}} < 63$
    $\mathrm{Head}_{N,1} \leftarrow \mathrm{Head}_{N,1} + \|\mathrm{pk}_N\|_{\mathrm{nib}}$
    return $\mathrm{Head}_N$
$\mathrm{Head}_{N,1} \leftarrow \mathrm{Head}_{N,1} + 63$
$l \leftarrow \|\mathrm{pk}_N\|_{\mathrm{nib}} - 62$
$i \leftarrow 2$
while $(l > 255)$
    $\mathrm{Head}_{N,i} \leftarrow 255$
    $l \leftarrow l - 255$
    $i \leftarrow i + 1$
$\mathrm{Head}_{N,i} \leftarrow l - 1$
return $\mathrm{Head}_N$
```

5.4 The Merkle proof

To prove the consistency of the state storage across the network and its modifications both efficiently and effectively, the Trie implements a Merkle tree structure. The hash value corresponding to each node needs to be computed rigorously to make the inter-implementation data integrity possible.

The Merkle value of each node should depend on the Merkle value of all its children as well as on its corresponding data in the state storage. This recursive dependancy is encompassed into the subvalue part of the node value which recursively depends on the Merkle value of its children.

We use the auxilary function introduced in Definition 26 to encode and decode information stored in a branch node.

Definition 26. *Suppose $N_b, N_c \in \mathcal{N}$ and N_c is a child of N_b. We define where bit $b_i \colon = 1$ if N has a child with partial key i, therefore we define* ***ChildrenBitmap*** *functions as follows:*

$$\begin{aligned}\text{ChildrenBitmap:}\ & \mathcal{N}_b \rightarrow \mathbb{B}_2 \\ & N \mapsto (b_{15}, \ldots, b_8, b_7, \ldots b_0)_2\end{aligned}$$

where

$$b_i := \begin{cases} 1 & \exists N_c \in \mathcal{N} \colon k_{N_c} = k_{N_b} || i || \text{pk}_{N_c} \\ 0 & \textit{otherwise} \end{cases}$$

Definition 27. *For a given node N, the* ***subvalue*** *of N, formally referred to as sv_N, is determined as follows: in a case which:*

$$\text{sv}_N := \begin{cases} \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & N \textit{ is a leaf node} \\ \text{ChildrenBitmap}(N) \| H(N_{C_1}) \ldots H(N_{C_n}) || \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & N \textit{ is a branch node} \end{cases}$$

Where $N_{C_1} \ldots N_{C_n}$ with $n \leqslant 16$ are the children nodes of the branch node N and Enc_{SC}, StoredValue, H, and ChildrenBitmap(N) are defined in Definitions ?,17, 28 and 26 respectively.

The Trie deviates from a traditional Merkle tree where node value, v_N (see Definition 24) is presented instead of its hash if it occupies less space than its hash.

Definition 28. *For a given node N, the* ***Merkle value*** *of N, denoted by $H(N)$ is defined as follows:*

$$H \colon \mathbb{B} \rightarrow \mathbb{B}_{32}$$

$$H(N) \colon \begin{cases} v_N || 0_{B_{32 - \| v_N \|}} & \| v_N \| < 32 \\ \text{Blake2}b(v_N) & \| v_N \| \geqslant 32 \end{cases}$$

Where $0_{32 - \| v_N \|}$ an all zero byte array of length $32 - || v_N ||$.

6 Transactions

6.1 Preliminaries

Definition 29. ***Account key*** **$(\mathbf{sk}^a, \mathbf{pk}^a)$** *is a pair of Ristretto SR25519 used to sign transactions among other accounts and blance-related functions.*

7 Consensus Engine

Consensus in Polkadot RE is achieved during the execution of two different procedures. The first procedure is block production and the second is finality. Polkadot RE must run these procedures, if and only if it is running on a validator node.

7.1 Block Production

Polkadot RE uses BABE protocol [Gro19] for block production designed based on Ouroboros praos [DGKR18]. BABE execution happens in sequential non-overlapping phases known as an ***epoch***. Each epoch on its turn is divided into a predefined number of slots. All slots in each epoch are sequentially indexed starting from 0. At the beginning of each epoch, the BABE node needs to run Algorithm 5 to find out in which slots it should produce a block and gossip to the other block producers. In turn, the block producer node should keep a copy of the block tree and grow it as it receives valid blocks from other block producers. A block producer prunes the tree in parallel using Algorithm ?.

7.1.1 Preliminaries

Definition 30. *A* ***block producer****, noted by $\mathcal{P}_j$, is a node running Polkadot RE which is authorized to keep a transaction queue and which gets a turn in producing blocks.*

Definition 31. ***Block authring session key pair*** $(\mathbf{sk}_j^s, \mathbf{pk}_j^s)$ *is an SR25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (see Definition 29) and is used to sign the produced block as well as to compute its lottery values in Algorithm 5.*

Definition 32. *A block production* ***epoch****, formally referred to as $\mathcal{E}$ is a period with pre-known starting time and fixed length during which the set of block producers stays constant. Epochs are indexed sequentially, and we refer to the n^{th} epoch since genesis by $\mathcal{E}_n$. Each epoch is divided into equal length periods known as block production* ***slots****, sequentially indexed in each epoch. Each slot is awarded to a subset of block producers during which they are allowed to generate a block.*

Notation 33. *We refer to the number of slots in epoch $\mathcal{E}_n$ by sc_n. sc_n is set to the result of calling runtime entry* `BabeApi_slot_duration` *at the “beginning of each epoch. For a given block B, we use the notation $\boldsymbol{s_B}$ to refer to the slot during which B has been produced. Conversely, for slot s, $\mathcal{B}_s$ is the set of Blocks generated at slot s.*

Definition 34 provides an iterator over the blocks produced during an specific epoch.

Definition 34. *By $\textsc{SubChain}(\mathcal{E}_n)$ for epoch $\mathcal{E}_n$, we refer to the path graph of BT which contains all the blocks generated during the slots of epoch $\mathcal{E}_n$. When there is more than one block generated at a slot, we choose the one which is also on $\textsc{Longest-Branch}(\mathrm{BT})$.*

7.1.2 Block Production Lottery

Definition 35. ***Winning threshold*** *denoted by $\boldsymbol{\tau}$ is the threshold which is used alongside with the result of Algorirthm 5 to decide if a block producer is the winner of a specific slot. τ is set to result of call into* `BabeApi_slot_winning_threshold` *runtime entry.*

A block producer aiming to produce a block during $\mathcal{E}_n$ should run Algorithm 5 to identify the slots it is awarded. These are the slots during which the block producer is allowed to build a block. The sk is the block producer lottery secret key and n is the index of epoch for whose slots the block producer is running the lottery.

Algorithm 5. Block-production-lottery(sk, n)

```
$r \leftarrow$ Epoch-Randomness($n$)
for $i := 1$ to $\mathrm{sc}_n$
    $(d, \pi) \leftarrow \mathrm{VRF}(r, i, \mathrm{sk})$
    $A[i] \leftarrow (d, \pi)$
return A
```

For any slot i in epoch n where $d < \tau$, the block producer is required to produce a block. For the definitions of Epoch-Randomness and VRF functions, see Algorithm 8 and Section 8.4 respectively.

7.1.3 Slot number calculation

It is essential for a block producer to calculate and validate the slot number at a certain point in time. Slots are dividing the time continuum in an overlapping interval. At a given time, the block producer should be able to determine the set of slots which can be associated to a valid block generated at that time. We formalize the notion of validity in the following definitions:

Definition 36. *The* ***slot tail****, formally referred to by* SlTl *represents the number of on-chain blocks that are used to estimate the slot time of a given slot. This number is set to be 1200.*

Algorithm 6 determines the slot time for a future slot based on the *block arrival time* associated with blocks in the slot tail defined in Definition 37.

Definition 37. *The* ***block arrival time*** *of block B for node j formally represented by $\boldsymbol{T_B^j}$ is the local time of node j when node j has received the block B for the first time. If the node j itself is the producer of B, T_B^j is set equal to the time that the block is produced. The index j in T_B^j notation may be dropped when there is no ambiguity about the underlying node.*

In addition to the arrival time of block B, the block producer also needs to know how many slots have passed since the arrival of B. This value is formalized in Definition 38.

Definition 38. *Let s_i and s_j be two slots belonging to epochs $\mathcal{E}_k$ and $\mathcal{E}_l$. By* Slot-Offset(s_i, s_j) *we refer to the function whose value is equal to the number of slots between s_i and s_j (counting s_j) on time continuum. As such, we have* Slot-Offset$(s_i, s_i) = 0$.

Algorithm 6. Slot-Time(s: the number of the slots whose time needs to be determined)

```
$T_s \leftarrow \{\}$
$B_d \leftarrow$ Deepest-Leaf(BT)
for $B_i$ in SubChain($B_{H_n(B_d)-\textsc{sitl}}$, $B_d$)
    $s_t^{B_i} \leftarrow T_{B_i} +$ Slot-Offset$(s_{B_i}, s) \times \mathcal{T}$
    $T_s \leftarrow T_s \cup s_t^{B_i}$
return Median($T_s$)
```

7.1.4 Block Production

At each epoch, each block producer should run Algorithm 7 to produce blocks during the slots it has been awarded during that epoch. The produced blocks need to be broadcasted alongside with the *babe header* defined in Definition 39.

Definition 39. *The* ***Babe Header*** *of block B, referred to formally by* $\boldsymbol{H_{\mathrm{Babe}}(B)}$ *is a tuple that consists of the following components:*

$$(\pi, S_B, \mathrm{pk}, s, d)$$

in which:

s*: is the slot at which the block is produced.*
π, d*: are the results of the block lottery for slot s.*
pk_j^s*: is the SR25519 session public key associated with the block producer.*
S_B*:* $\mathrm{Sig}_{\mathrm{SR25519},\mathrm{sk}_j^s}(\mathrm{Enc}_{\mathrm{SC}}(s, \mathrm{Black2}s(\mathrm{Head}(B), \pi)))$

Algorithm 7. INVOKE-BLOCK-AUTHORING(sk, pk, n, BT: Current Block Tree)

```
$A \leftarrow$ BLOCK-PRODUCTION-LOTTERY(sk, $n$)
for $s \leftarrow 1$ to $\mathrm{sc}_n$
    WAIT(until SLOT-TIME(s))
    $(d, \pi) \leftarrow A[s]$
    if $d < \tau$
        $C_{\mathrm{Best}} \leftarrow$ LONGEST-BRANCH(BT)
        $B_s \leftarrow$ BUILD-BLOCK($C_{\mathrm{Best}}$)
        BROADCAST-BLOCK($B_s, H_{\mathrm{Babe}}(B_s)$)
```

7.1.5 Block Validation

7.1.6 Epoch Randomness

At the end of epoch $\mathcal{E}_n$, each block producer is able to compute the randomness seed it needs in order to participate in the block production lottery in epoch $\mathcal{E}_{n+2}$. The computation of the seed is described in Algorithm 8 which uses the concept of epoch subchain described in Definition 34.

Algorithm 8. EPOCH-RANDOMNESS($n > 2$: epoch index)

```
$\rho \leftarrow \phi$
for $B$ in SUBCHAIN($\mathcal{E}_{n-2}$)
    $\rho \leftarrow \rho || d_B$
return Blake2b(EPOCH-RANDOMNESS($n - 1$)$||n||\rho$)
```

In which value d_B is the VRF output computed for slot s_B by running Algorithm 5.

7.2 Finality

Polkadot RE uses GRANDPA Finality protocol [Ali19] to finalize blocks. Finality is obtained by consecutive rounds of voting by validator nodes. Validators execute GRANDPA finality process in parallel to Block Production as an independent service. In this section, we describe the different functions that GRANDPA service is supposed to perform to successfully participate in the block finalization process.

7.2.1 Preliminaries

Definition 40. *A **GRANDPA Voter**, v, is represented by a key pair $(k_v^{\text{pr}}, v_{\text{id}})$ where k_v^{pr} represents its private key which is an* ED25519 *private key, is a node running GRANDPA protocol, and broadcasts votes to finalize blocks in a Polkadot RE - based chain. The **set of all GRANDPA voters** is indicated by $\mathbb{V}$. For a given block B, we have4*

$$\mathbb{V}_B = \texttt{authorities}(B)$$

where `authorities` *is the entry into runtime described in Section 3.3.2.*

Definition 41. ***GRANDPA state**,* GS*, is defined as*

$$\text{GS} := \{\mathbb{V}, \text{id}_{\mathbb{V}}, r\}$$

where:

$\mathbb{V}$*: is the set of voters.*

$\mathbb{V}_{\mathbf{id}}$*: is an incremental counter tracking membership, which changes in V.*

$\boldsymbol{r}$*: is the voting round number.*

Now we need to define how Polkadot RE counts the number of votes for block B. First a vote is defined as:

Definition 42. *A **GRANDPA vote** or simply a vote for block B is an ordered pair defined as*

$$V(B) := (H_h(B), H_i(B))$$

where $H_h(B)$ and $H_i(B)$ are the block hash and the block number defined in Definitions 13 and 14 respectively.

Definition 43. *Voters engage in a maximum of two sub-rounds of voting for each round r. The first sub-round is called **pre-vote** and the second sub-round is called **pre-commit**.*

By $\boldsymbol{V_v^{r,\mathbf{pv}}}$ and $\boldsymbol{V_v^{r,\mathbf{pc}}}$ we refer to the vote cast by voter v in round r (for block B) during the pre-vote and the pre-commit sub-round respectively.

The GRANDPA protocol dictates how an honest voter should vote in each sub-round, which is described in Algorithm 10. After defining what constitues a vote in GRANDPA, we define how GRANDPA counts votes.

Definition 44. *Voter v **equivocates** if they broadcast two or more valid votes to blocks not residing on the same branch of the block tree during one voting sub-round. In such a situation, we say that v is an **equivocator** and any vote $V_v^{r,\text{stage}}(B)$ cast by v in that round is an **equivocatory vote** and*

$$\mathcal{E}^{r,\text{stage}}$$

represents the set of all equivocators voters in sub-round “stage” *of round r. When we want to refer to the number of equivocators whose equivocation has been observed by voter v we refer to it by:*

$$\mathcal{E}_{\text{obs}(v)}^{r,\text{stage}}$$

Definition 45. *A vote $V_v^{r,\text{stage}} = V(B)$ is **invalid** if*

- *$H(B)$ does not correspond to a valid block;*
- *B is not an (eventual) descendant of a previously finalized block;*
- *$M_v^{r,\text{stage}}$ does not bear a valid signature;*
- *$\text{id}_{\mathbb{V}}$ does not match the current $\mathbb{V}$;*
- *If $V_v^{r,\text{stage}}$ is an equivocatory vote.*

Definition 46. *For validator v, **the set of observed direct votes for Block B in round r**, formally denoted by $\text{VD}_{\text{obs}(v)}^{r,\text{stage}}(B)$ is equal to the union of:*

- *set of valid votes $V_{v_i}^{r,\text{stage}}$ cast in round r and received by v such that $V_{v_i}^{r,\text{stage}} = V(B)$.*

Definition 47. *We refer to **the set of total votes observed by voter v in sub-round “stage” of round r** by $\boldsymbol{V_{\text{obs}(v)}^{r,\text{stage}}}$.*

*The **set of all observed votes by v in the sub-round stage of round r for block B**, $\boldsymbol{V_{\text{obs}(v)}^{r,\text{stage}}(B)}$ is equal to all of the observed direct votes casted for block B and all of the B's descendents defined formally as:*

$$V_{\text{obs}(v)}^{r,\text{stage}}(B) := \bigcup_{v_i \in \mathbb{V}, B \geqslant B'} \text{VD}_{\text{obs}(v)}^{r,\text{stage}}(B')$$

*The **total number of observed votes for Block B in round r** is defined to be the size of that set plus the total number of equivocators voters:*

$$\# V_{\text{obs}(v)}^{r,\text{stage}}(B) = |V_{\text{obs}(v)}^{r,\text{stage}}(B)| + |\mathcal{E}_{\text{obs}(v)}^{r,\text{stage}}|$$

Definition 48. *The current **pre-voted** block $B_v^{r,\text{pv}}$ is the block with*

$$H_n(B_v^{r,\text{pv}}) = \text{Max}(H_n(B) | \forall B : \# V_{\text{obs}(v)}^{r,\text{pv}}(B) \geqslant 2/3 |\mathbb{V}|)$$

Note that for genesis block Genesis we always have $\# V_{\text{obs}(v)}^{r,\text{pv}}(B) = |\mathbb{V}|$.

Finally, we define when a voter v see a round as completable, that is when they are confident that $B_v^{r,\text{pv}}$ is an upper bound for what is going to be finalised in this round.

Definition 49. *We say that round r is **completable** if $|V_{\text{obs}(v)}^{r,\text{pc}}| + \mathcal{E}_{\text{obs}(v)}^{r,\text{pc}} > \frac{2}{3}\mathbb{V}$ and for all $B' > B_v^{r,\text{pv}}$:*

$$|V_{\text{obs}(v)}^{r,\text{pc}}| - \mathcal{E}_{\text{obs}(v)}^{r,\text{pc}} - |V_{\text{obs}(v)}^{r,\text{pc}}(B')| > \frac{2}{3}|\mathbb{V}|$$

Note that in practice we only need to check the inequality for those $B' > B_v^{r,\text{pv}}$ where $|V_{\text{obs}(v)}^{r,\text{pc}}(B')| > 0$.

7.2.2 Voting Messages Specification

Voting is done by means of broadcasting voting messages to the network. Validators inform their peers about the block finalized in round r by broadcasting a finalization message (see Algorithm 10 for more details). These messages are specified in this section.

Definition 50. *A vote casted by voter v should be broadcasted as a* ***message $M_v^{r,\text{stage}}$*** *to the network by voter v with the following structure:*

$$M_v^{r,\text{stage}} := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}}, \text{Enc}_{\text{SC}}(\text{stage}, V_v^{r,\text{stage}}, \text{Sig}_{\text{ED25519}}(\text{Enc}_{\text{SC}}(\text{stage}, V_v^{r,\text{stage}}, r, V_{\text{id}}), v_{\text{id}})$$

Where:

r*:*	*round number*	*64 bit integer*
V_{id}*:*	*incremental change tracker counter*	*64 bit integer*
v_{id}*:*	*Ed25519 public key of v*	*4 byte array*
stage*:*	*0 if it is the pre-vote sub-round* *1 if it the pre-commit sub-round*	*1 byte*

Definition 51. *The* ***justification for block B in round r*** *of GRANDPA protocol defined $J^r(B)$ is a vector of pairs of the type:*

$$(V(B'), (\text{Sign}_{v_i}^{r,\text{pc}}(B'), v_{\text{id}}))$$

in which either

$$B' > B$$

or $V_{v_i}^{r,\text{pc}}(B')$ is an equivocatory vote.

In all cases, $\text{Sign}_{v_i}^{r,\text{pc}}(B')$ is the signature of voter v_i broadcasted during the pre-commit sub-round of round r.

Definition 52. GRANDPA ***finalizing message for block B in round r*** *represented as* ***$M_v^{r,\text{Fin}}$(B)*** *is a message broadcasted by voter v to the network indicating that voter v has finalized block B in round r. It has the following structure:*

$$M_v^{r,\text{Fin}}(B) := \text{Enc}_{\text{SC}}(r, V(B), J^r(B))$$

in which $J^r(B)$ in the justification defined in Definition 51.

7.2.3 Initiating the GRANDPA State

A validator needs to initiate its state and sync it with other validators, to be able to participate coherently in the voting process. In particular, considering that voting is happening in different rounds and each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r in accordance with the current voting round r_n, which is currently undergoing in the network.

As instructed in Algorithm 9, whenever the membership of GRANDPA voters changes, r is set to 0 and V_{id} needs to be incremented.

Algorithm 9. Join-Leave-Grandpa-Voters ($\mathcal{V}$)

```
$r \leftarrow 0$
$\mathcal{V}_{\text{id}} \leftarrow \text{ReadState}('\text{AUTHORITY\_SET\_KEY}')$
$\mathcal{V}_{\text{id}} \leftarrow \mathcal{V}_{\text{id}} + 1$
Execute-One-Grandpa-Round($r$)
```

7.2.4 Voting Process in Round r

For each round r, an honest voter v must participate in the voting process by following Algorithm 10.

Algorithm 10. Play-Grandpa-round(r)

```
$t_{r,v} \leftarrow$ Time
primary $\leftarrow$ Derive-Primary
if $v =$ primary
    Broadcast($M_v^{r-1,\mathrm{Fin}}$(Best-Final-Candidate($r$-1)))
Receive-Messages(until Time $\geqslant t_{r,v} + 2 \times T$ or $r$ is completable)
$L \leftarrow$ Best-Final-Candidate($r$-1)
if Received($M_{v_{\mathrm{primary}}}^{r,\mathrm{pv}}(B)$) and $B_v^{r,\mathrm{pv}} \geqslant B > L$
    $N \leftarrow B$
else
    $N \leftarrow B' : H_n(B') = \max \{H_n(B') : B' > L\}$
Broadcast($M_v^{r,\mathrm{pv}}(N)$)
Receive-Messages(until $B_v^{r,\mathrm{pv}} \geqslant L$ and (Time $\geqslant t_{r,v} + 4 \times T$ or $r$ is completable))
Broadcast($M_v^{r,\mathrm{pc}}(B_v^{r,\mathrm{pv}})$)
Play-Grandpa-round($r + 1$)
```

The condition of *completablitiy* is defined in Definition 49. Best-Final-Candidate function is explained in Algorithm 11.

Algorithm 11. Best-Final-Candidate(r)

```
$\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,\mathrm{pv}} : |V_v^{r,\mathrm{pc}}| - \#V_v^{r,\mathrm{pc}}(B') \leqslant 1/3|\mathbb{V}|\}$
if $\mathcal{C} = \phi$
    return $\phi$
else
    return $E \in \mathcal{C} : H_n(E) = \max \{H_n(B') : B' \in \mathcal{C}\}$
```

Algorithm 12. Attempt-To-Finalize-Round(r)

```
$L \leftarrow$ Last-Finalized-Block
$E \leftarrow$ Best-Final-Candidate($r$)
if $E \geqslant L$ and $V_{\mathrm{obs}(v)}^{r-1,\mathrm{pc}}(E) > 2/3|\mathcal{V}|$
    Last-Finalized-Block$\leftarrow B^{r,\mathrm{pc}}$
    if $M_v^{r,\mathrm{Fin}}(E) \notin$ Received-Messages
        Broadcast($M_v^{r,\mathrm{Fin}}(E)$)
```

```
            return
schedule-call Attempt-To-Finalize-Round($r$) when Receive-Messages
```

8 Cryptographic Algorithms

8.1 Hash functions

8.2 BLAKE2

BLAKE2 is a collection of cryptographic hash functions known for their high speed. their design closely resembles BLAKE which has been a finalist in SHA-3 competition.

Polkadot is using Blake2b variant which is optimized for 64bit platforms. Unless otherwise specified, Blake2b hash function with 256bit output is used whenever Blake2b is invoked in this document. The detailed specification and sample implementations of all variants of Blake2 hash functions can be found in RFC 7693 [SA15].

8.3 Randomness

8.4 VRF

9 Auxiliary Encodings

9.1 SCALE Codec

Polkadot RE uses *Simple Concatenated Aggregate Little-Endian" (SCALE) codec* to encode byte arrays as well as other data structures. SCALE provides a canonical encoding to produce consistent hash values across their implementation, including the Merkle hash proof for the State Storage.

Definition 53. *The* ***SCALE codec*** *for* ***Byte array*** *A such that*

$$A := b_1 b_2 \ldots b_n$$

such that $n < 2^{536}$ is a byte array refered to $\mathrm{Enc}_{\mathrm{SC}}(A)$ and defined as:

$$\mathrm{Enc}_{\mathrm{SC}}(A) := \mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}(\|A\|)||A$$

where $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$ is defined in Definition 58.

Definition 54. *The* ***SCALE codec*** *for* ***Tuple*** *T such that:*

$$T := (A_1, \ldots, A_n)$$

*Where A_i's are values of **different types**, is defined as:*

$$\mathrm{Enc}_{\mathrm{SC}}(T) := \mathrm{Enc}_{\mathrm{SC}}(A_1)|\mathrm{Enc}_{\mathrm{SC}}(A_2)|...|\,\mathrm{Enc}_{\mathrm{SC}}(A_n)$$

In case of a tuple (or struct), the knowledge of the shape of data is not encoded even though it is necessary for decoding. The decoder needs to derive that information from the context where the encoding/decoding is happenning.

Definition 55. *The **SCALE codec** for **sequence** S such that:*

$$S := A_1, ..., A_n$$

*where A_i's are values of **the same type** (and the decoder is unable to infer value of n from the context) is defined as:*

$$\mathrm{Enc}_{\mathrm{SC}}(T) := \mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}(\|S\|)\mathrm{Enc}_{\mathrm{SC}}(A_1)|\mathrm{Enc}_{\mathrm{SC}}(A_2)|...|\,\mathrm{Enc}_{\mathrm{SC}}(A_n)$$

where $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$ *is defined in Definition 58.*

Definition 56. *The **SCALE codec** for **boolean value** b defined as a byte as follows:*

$$\begin{array}{rl} \mathrm{Enc}_{\mathrm{SC}}\colon & \{\mathrm{False}, \mathrm{True}\} \rightarrow \mathbb{B}_1 \\ b \rightarrow & \begin{cases} 0 & b = \mathrm{False} \\ 1 & b = \mathrm{True} \end{cases} \end{array}$$

Definition 57. *The **SCALE codec,** $\mathbf{Enc_{SC}}$ for other types such as fixed length integers not defined here otherwise, is equal to little endian encoding of those values defined in Definition 7.*

9.1.1 Length Encoding

SCALE Length encoding is used to encode integer numbers of variying sizes prominently in an encoding length of arrays:

Definition 58. ***SCALE Length Encoding,*** $\mathbf{Enc_{SC}^{Len}}$ *also known as compact encoding of a non-negative integer number n is defined as follows:*

$$\begin{array}{rl} \mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}\colon & \mathbb{N} \rightarrow \mathbb{B} \\ n \rightarrow b := & \begin{cases} l_1 & 0 \leqslant n < 2^6 \\ i_1\, i_2 & 2^6 \leqslant n < 2^{14} \\ j_1\, j_2\, j_3 & 2^{14} \leqslant n < 2^{30} \\ k_1\, k_2 \ldots k_m & 2^{30} \leqslant n \end{cases} \end{array}$$

in where the bits of byte array b are defined as follows:

$$\begin{array}{rcl} l_1^1\, l_1^0 & = & 00 \\ i_1^1\, i_1^0 & = & 01 \\ j_1^1\, j_1^0 & = & 10 \\ k_1^1\, k_1^0 & = & 11 \end{array}$$

and the rest of the bits of b store the value of n in little-endian format in base-2 as follows:

$$\left.\begin{array}{ll} l_1^7 \ldots l_1^3\, l_1^2 & n < 2^6 \\ i_2^7 \ldots i_2^0\, i_1^7 \ldots i_1^2 & 2^6 \leqslant n < 2^{14} \\ j_4^7 \ldots j_4^0\, j_3^7 \ldots j_1^7 \ldots j_1^2 & 2^{14} \leqslant n < 2^{30} \\ k_2 + k_3\, 2^8 + k_4\, 2^{2 \cdot 8} + \cdots + k_m\, 2^{(m-2)8} & 2^{30} \leqslant n \end{array}\right\} := n$$

such that:

$$k_1^7 \ldots k_1^3\, k_1^2 \colon = m - 4$$

9.2 Hex Encoding

Practically, it is more convenient and efficient to store and process data which is stored in a byte array. On the other hand, the Trie keys are broken into 4-bits nibbles. Accordingly, we need a method to encode sequences of 4-bits nibbles into byte arrays canonically:

Definition 59. *Suppose that* $\mathrm{PK} = (k_1, \ldots, k_n)$ *is a sequence of nibbles, then*
$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) :=$

$$\left\{\begin{array}{lll} \mathrm{Nibbles}_4 & \rightarrow & \mathbb{B} \\ \mathrm{PK} = (k_1, \ldots, k_n) & \mapsto & \left\{\begin{array}{ll} (16k_1 + k_2, \ldots, 16k_{2i-1} + k_{2i}) & n = 2\,i \\ (k_1, 16k_2 + k_3, \ldots, 16k_{2i} + k_{2i+1}) & n = 2\,i + 1 \end{array}\right. \end{array}\right.$$

10 Genesis Block Specification

11 Predefined Storage keys

12 Runtime upgrade

Appendix A Runtime API

Runtime API is a set of functions that Polkadot RE exposes to Runtime to access external functions needed for various reasons, such as Storage of content, access and manipulation, memory allocation, and also efficiency. The functions are specified in each subsequent subsection for each category of those functions.

A.1 Storage

A.1.1 `ext_set_storage`

Sets the value of a specific key in the state storage.

Prototype:

```
(func $ext_storage
  (param $key_data i32) (param $key_len i32) (param $value_data i32)
  (param $value_len i32))
```

Arguments:

- `key`: a pointer indicating the buffer containing the key.
- `key_len`: the key length in bytes.
- `value`: a pointer indicating the buffer containing the value to be stored under the key.
- `value_len`: the length of the value buffer in bytes.

A.1.2 `ext_storage_root`

Retrieves the root of the state storage.

Prototype:

```
(func $ext_storage_root
  (param $result_ptr i32))
```

Arguments:

- `result_ptr`: a memory address pointing at a byte array which contains the root of the state storage after the function concludes.

A.1.3 `ext_blake2_256_enumerated_trie_root`

Given an array of byte arrays, arranges them in a Merkle trie, defined in Section 5.4, and computes the trie root hash.

Prototype:

```
(func $ext_blake2_256_enumerated_trie_root
      (param $values_data i32) (param $lens_data i32) (param $lens_len i32)
      (param $result i32))
```

Arguments:

- `values_data`: a memory address pointing at the buffer containing the array where byte arrays are stored consecutively.
- `lens_data`: an array of `i32` elements each stores the length of each byte array stored in `value_data`.
- `lens`_len: the number of `i32` elements in `lens_data`.
- `result`: a memory address pointing at the beginning of a 32-byte byte array containing the root of the Merkle trie corresponding to elements of `values_data`.

A.1.4 `ext_clear_prefix`

Given a byte array, this function removes all storage entries whose key matches the prefix specified in the array.

Prototype:

```
(func $ext_clear_prefix
      (param $prefix_data i32) (param $prefix_len i32))
```

Arguments:

- `prefix_data`: a memory address pointing at the buffer containing the byte array containing the prefix.
- `prefix_len`: the length of the byte array in number of bytes.

A.1.5 `ext_clear_storage`

Given a byte array, this function removes the storage entry whose key is specified in the array.

Prototype:

```
(func $ext_clear_storage
      (param $key_data i32) (param $key_len i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.

A.1.6 `ext_exists_storage`

Given a byte array, this function checks if the storage entry corresponding to the key specified in the array exists.

Prototype:

```
(func $ext_exists_storage
      (param $key_data i32) (param $key_len i32) (result i32)
    )
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `result`: An `i32` integer which is equal to 1 verifies if an entry with the given key exists in the storage or 0 if the key storage does not contain an entry with the given key.

A.1.7 `ext_get_allocated_storage`

Given a byte array, this function allocates a large enough buffer in the memory and retrieves the value stored under the key that is specified in the array. Then, it stores it in the allocated buffer if the entry exists in the storage.

Prototype:

```
    (func $get_allocated_storage
      (param $key_data i32) (param $key_len i32) (param $written_out i32) (result i32))
```

Arguments:

- key_data: a memory address pointing at the buffer containing the byte array containing the key value.
- key_len: the length of the byte array in number of bytes.
- written_out: the function stores the length of the retrieved value in number of bytes if the enty exists. If the entry does not exist, it returns $2^{32}-1$.
- result: A pointer to the buffer in which the function allocates and stores the value corresponding to the given key if such an entry exist; otherwise it is equal to 0.

A.1.8 ext_get_storage_into

Given a byte array, this function retrieves the value stored under the key specified in the array and stores a specified chunk of it in the provided buffer, if the entry exists in the storage.

Prototype:

```
(func $ext_get_storage_into
  (param $key_data i32) (param $key_len i32) (param $value_data i32)
  (param $value_len i32) (param $value_offset i32) (result i32))
```

Arguments:

- key_data: a memory address pointing at the buffer containing the byte array containing the key value.
- key_len: the length of the byte array in number of bytes.
- value_data: a pointer to the buffer in which the function stores the chunk of the value it retrieves.
- value_len: the (maximum) length of the chunk in bytes the function will read of the value and will store in the value_data buffer.
- value_offset: the offset of the chunk where the function should start storing the value in the provided buffer, i.e. the number of bytes the functions should skip from the retrieved value before storing the data in the value_data in number of bytes.
- result: The number of bytes the function writes in value_data if the value exists or $2^{32}-1$ if the entry does not exist under the specified key.

A.1.9 To be Specced

- ext_clear_child_storage
- ext_exists_child_storage
- ext_get_allocated_child_storage
- ext_get_child_storage_into
- ext_kill_child_storage
- ext_set_child_storage
- ext_storage_changes_root

A.2 Memory

A.2.1 ext_malloc

Allocates memory of a requested size in the heap.

Prototype:

```
(func $ext_malloc
  (param $size i32) (result i32))
```

Arguments:

- `size`: the size of the buffer to be allocated in number of bytes.

Result:

a memory address pointing at the beginning of the allocated buffer.

A.2.2 ext_free

Deallocates a previously allocated memory.

Prototype:

```
(func $ext_free
      (param $addr i32))
```

Arguments:

- `addr`: a 32bit memory address pointing at the allocated memory.

A.2.3 Input/Output

- `ext_print_hex`
- `ext_print_num`
- `ext_print_utf8`

A.3 Cryptograhpic auxiliary functions

A.3.1 ext_blake2_256

Computes the Blake2b 256bit hash of a given byte array.

Prototype:

```
(func (export "ext_blake2_256")
      (param $data i32) (param  $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.

- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 32-byte byte array contanining the Blake2b hash of the data.

A.3.2 `ext_keccak_256`

Computes the Keccak-256 hash of a given byte array.

Prototype:

```
(func $ext_keccak_256
      (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 32-byte byte array contanining the Keccak-256 hash of the data.

A.3.3 `ext_twox_128`

Computes the *xxHash64* algorithm (see [Col19]) twice initiated with seeds 0 and 1 and applied on a given byte array and outputs the concatenated result.

Prototype:

```
(func $ext_twox_128
       (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 16-byte byte array containing $xxhash64_0(\texttt{data})||xxhash64_1(\texttt{data})$ where $xxhash64_i$ is the xxhash64 function initiated with seed i as a 64bit unsigned integer.

A.3.4 `ext_ed25519_verify`

Given a message signed by the ED25519 signature algorithm alongside with its signature and the allegedly signer public key, it verifies the validity of the signature by the provided public key.

Prototype:

```
(func $ext_ed25519_verify
      (param $msg_data i32) (param $msg_len i32) (param $sig_data i32)
      (param $pubkey_data i32) (result i32))
```

Arguments:

- `msg_data`: a pointer to the buffer containing the message body.

- msg_len: an i32 integer indicating the size of the message buffer in bytes.
- sig_data: a pointer to the 64 byte memory buffer containing the ED25519 signature corresponding to the message.
- pubkey_data: a pointer to the 32 byte buffer containing the public key and corresponding to the secret key which has signed the message.
- result: an integer value equal to 0 indicating the validity of the signature or a nonzero value otherwise.

A.3.5 ext_sr25519_verify

Given a message signed by the SR25519 signature algorithm alongside with its signature and the allegedly signer public key, it verifies the validity of the signature by the provided public key.

Prototype:

```
(func $ext_sr25519_verify
      (param $msg_data i32) (param $msg_len i32) (param $sig_data i32)
      (param $pubkey_data i32) (result i32))
```

Arguments:

- msg_data: a pointer to the buffer containing the message body.
- msg_len: an i32 integer indicating the size of the message buffer in bytes.
- sig_data: a pointer to the 64 byte memory buffer containing the SR25519 signature corresponding to the message.
- pubkey_data: a pointer to the 32 byte buffer containing the public key and corresponding to the secret key which has signed the message.
- result: an integer value equal to 0 indicating the validity of the signature or a nonzero value otherwise.

A.3.6 To be Specced

- ext_twox_256

A.4 Sandboxing

A.4.1 To be Specced

- ext_sandbox_instance_teardown
- ext_sandbox_instantiate
- ext_sandbox_invoke
- ext_sandbox_memory_get
- ext_sandbox_memory_new
- ext_sandbox_memory_set
- ext_sandbox_memory_teardown

A.5 Auxillary Debugging API

A.5.1 ext_print_hex

Prints out the content of the given buffer on the host's debugging console. Each byte is represented as a two-digit hexadecimal number.

Prototype:

```
(func $ext_print_hex
  (param $data i32) (parm $len i32))
```

Arguments:

- `data`: a pointer to the buffer containing the data that needs to be printed.
- `len`: an `i32` integer indicating the size of the buffer containing the data in bytes.

A.5.2 ext_print_utf8

Prints out the content of the given buffer on the host's debugging console. The buffer content is interpreted as a UTF-8 string if it represents a valid UTF-8 string, otherwise does nothing and returns.

Prototype:o

```
(func $ext_print_utf8
  (param $utf8_data i32) (param $utf8_len i32))
```

Arguments:

- `utf8_data`: a pointer to the buffer containing the utf8-encoded string to be printed.
- `utf8_len`: an `i32` integer indicating the size of the buffer containing the UTF-8 string in bytes.

A.6 Misc

A.6.1 To be Specced

- `ext_chain_id`

A.7 Not implemented in Polkadot-JS

Bibliography

[Ali19] Alistair Stewart. GRANDPA: A Byzantine Finality Gadgets, 2019.

[Col19] Yann Collet. Extremely fast non-cryptographic hash algorithm. Technical report, -, http://cyan4973.github.io/xxHash/, 2019.

[DGKR18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[Gro19] W3F Research Group. Blind Assignment for Blockchain Extension. Technical Specification, Web 3.0 Foundation, http://research.web3.foundation/en/latest/polkadot/BABE/Babe/, 2019.

[SA15] Markku Juhani Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). RFC 7693, https://tools.ietf.org/html/rfc7693, 2015.