Polkadot Runtime

Protocol Specification

Contents

Chapter 1

Extrinsics

1.1 Introduction

An extrinsic is a SCALE encoded array consisting of a version number, signature, and varying data types indicating the resulting Runtime function to be called, including the parameters required for that function to be executed.

1.2 Preliminaries

Definition 1 *An extrinsic , tx, is a tuple consisting of the extrinsic version, T_v (Def. 2), and the body of the extrinsic, T_b.*

$$tx := (T_v, T_b)$$

The value of T_b varies for each version. The current version 4 is described in section 1.3.1.

Definition 2 *T_v is a single byte and defines the extrinsic version. The required format of an extrinsic is dictated by the Runtime. Older or unsupported version are rejected.*

T_v does not consist of the version number directly, rather the version number is manipulated with bitwise operators in order to indicate whether the extrinsic is signed ("transaction") or unsigned ("inherent").

*If the extrinsic is **signed**, and therefore a transaction, T_v is defined as:*

$$T_v := n \mid 1000\ 0000$$

where n is the version number and $\mid$ implies a bitwise OR operator.

*If the extrinsic is **unsigned**, and therefore an inherent, T_v is defined as:*

$$T_v := n \ \& \ 0111\ 1111$$

where n is the version number and & implies a bitwise AND operator.

As an example, for extrinsic format version 4, an signed extrinsic represents T_v as `132` *while a unsigned extrinsic represents it as* `4`*.*

1.3 Extrinsics Body

1.3.1 Version 4

Version 4 of the Polkadot extrinsic format is defined as follows:

$$T_b := (A_i, Sig, E, M_i, F_i(m))$$

where each values represents:

- A_i: the 32-byte address of the sender (Def. 3).
- Sig: the signature of the sender (Def. 4).
- E: the extra data for the extrinsic (Def. 5).
- M_i: the indicator of the Polkadot module (Def. 7).
- $F_i(m)$: the indicator of the function of the Polkadot module (Def. 8).

Definition 3 *Account Id, A_i, is the 32-byte address of the sender of the extrinsic as described in the external SS58 address format.*

Definition 4 *The signature, Sig, is a varying data type indicating the used signature type, followed by the signature created by the extrinsic author. The following types are supported:*

$$Sig := \begin{cases} 0, & \textit{Ed25519, followed by: } (b_0, ..., b_{63}) \\ 1, & \textit{Sr25519, followed by: } (b_0, ..., b_{63}) \\ 2, & \textit{Ecdsa, followed by: } (b_0, ..., b_{64}) \end{cases}$$

Signature types vary in sizes, but each individual type is always fixed-size and therefore does not contain a length prefix. `Ed25519` *and* `Sr25519` *signatures are 512-bit while* `Ecdsa` *is 560-bit, where the last 8 bits are the recovery ID.*

The signature is created by signing payload P.

$$P := \begin{cases} Raw, & \textit{if } |Raw| \leq 256 \\ Blake2(Raw), & \textit{if } |Raw| > 256 \end{cases} \tag{1.1}$$
$$Raw := (M_i, F_i(m), E, R_v, F_v, H_h(G), H_h(B))$$

where each value represents:

- M_i: *the module indicator (Def. 7).*
- $F_i(m)$: *the function indicator of the module (Def. 8).*
- E: *the extra data (Def. 5).*
- R_v: *a UINT32 containing the specification version of* `14`.
- F_v: *a UINT32 containing the format version of* `2`.
- $H_h(G)$: *a 32-byte array containing the genesis hash.*
- $H_h(B)$: *a 32-byte array containing the hash of the block which starts the mortality period, as described in Definition 6.*

Definition 5 *Extra data, E, is a tuple containing additional meta data about the extrinsic and the system it is meant to be executed in.*

$$E := (T_m, N, P_t)$$

where each value represents:

- T_m: *contains the SCALE encoded mortality of the extrinsic (Def. 6).*
- N: *a compact integer containing the nonce of the sender. The nonce must be incremented by one for each extrinsic created, otherwise the Polkadot network will reject the extrinsic.*
- P_t: *a compact integer containing the transactor pay including tip.*

Definition 6 *Extrinsic **mortality** is a mechanism which ensures that an extrinsic is only valid within a certain period of the ongoing Polkadot lifetime. Extrinsics can also be immortal, as clarified in Section 6.*

The mortality mechanism works with two related values:

- M_{per}: *the period of validity in terms of block numbers from the block hash specified as $H_h(B)$ in the payload (Def. 4). The period must be the power of two, such as* `32`, `64`, `128`, *etc.*
- M_{pha}: *the phase in the period that this extrinsic's lifetime begins. This value is calculated with a formula and validators can use this value in order to determine which block hash is included in the payload.*

The extrinsic author uses the number of the block which starts the mortality period, $H_i(B)$, and specifies the desired period in order the calculate the phase:

$$M_{pha} = H_i(B) \ mod \ M_{per}$$

M_{per} and M_{pha} are then included in the extrinsic, as clarified in Definition 5, in the SCALE encoded form of T_m (Sect. 6). Polkadot validators can use M_{pha} to figure out the block hash included in the payload, which will therefore result in a valid signature if the extrinsic is within the specified period or an invalid signature if the extrinsic "died".

Example

The extrinsic author choses $M_{per} = 256$ *at block* `10'000`*, resulting with* $M_{pha} = 16$*. The extrinsic is then valid for blocks ranging from* `10'000` *to* `10'256`*.*

Encoding

T_m refers to the SCALE encoded form of type M_{per} and M_{pha}. T_m is the size of two bytes if the extrinsic is considered mortal, or simply one bytes with the value equal to zero if the extrinsic is considered immortal.

$$T_m := Enc_{SC}(M_{per}, M_{pha})$$

The SCALE encoded representation of mortality T_m deviates from most other types, as it's specialized to be the smallest possible value, as described in Algorithm 1 and 2.

Algorithm 1 ENCODE MORTALITY

Input: M_{per}, M_{pha}

// If the extrinsic is immortal, specify
// a single byte with the value equal to zero.

return $\begin{cases} 0 & if\ extrinsic\ is\ immortal \end{cases}$

Init $factor = \text{MAX}(M_{per} >> 12, 1)$
Init $left = \text{MIN}(\text{MAX}(\text{TZ}(M_{per}) - 1,\ 1),\ 15)$
Init $right = \frac{M_{pha}}{factor} << 4$

// Returns a two byte value
return $left|right$

Algorithm 2 DECODE MORTALITY

Input: T_m

return $\begin{cases} Immortal & if\ T^{b0}_{mort} = 0 \end{cases}$

Init $enc = T^{b0}_{mort} + (T^{b1}_{mort} << 8)$
Init $M_{per} = 2 << (enc\ mod\ (1 << 4))$
Init $factor = \text{MAX}(M_{per} >> 12, 1)$
Init $M_{pha} = (enc >> 4) * factor$

return $\begin{cases} (M_{per}, M_{pha}) & if\ M_{per} \geq 4 \textbf{ and } M_{pha} < M_{per} \\ Error & else \end{cases}$

- T_{mort}^{b0}: *the first byte of* T_m.
- T_{mort}^{b1}: *the second byte of* T_m.
- MIN(..): *returns the minimum of two values.*
- MAX(..): *returns the maximum of two values.*
- TZ(num): *returns the number of trailing zeros in the binary representation of num. For example, the binary representation of* `40` *is* `0010 1000`*, which has three trailing zeros.*
- $>>$: *performs a binary right shift operation.*
- $<<$: *performs a binary left shift operation.*
- $|$: *performs a bitwise OR operation.*

Definition 7 *M_i is an indicator for the Runtime to which Polkadot module, m, the extrinsic should be forwarded to.*

M_i is a varying data type pointing to every module exposed to the network.

$$M_i := \begin{cases} 0, & System \\ 1, & Utility \\ ... & \\ 7, & Balances \\ ... & \end{cases}$$

Definition 8 *$F_i(m)$ is a tuple which contains an indicator, m_i, for the Runtime to which function within the Polkadot module, m, the extrinsic should be forwarded to. This indicator is followed by the concatenated and SCALE encoded parameters of the corresponding function, $params$.*

$$F_i(m) := (m_i, params)$$

The value of m_i varies for each Polkadot module, since every module offers different functions. As an example, the `Balances` *module has the following functions:*

$$Balances_i := \begin{cases} 0, & transfer \\ 1, & set\_balance \\ 2 & force\_transfer \\ 3 & transfer\_keep\_alive \end{cases}$$

Chapter 2

Availability and Validity Verification

2.1 Introduction

Validators are responsible for guaranteeing the validity and availability of PoV blocks. There are two phases of validation that takes place in the AnV protocol.

The primary validation check is carried out by parachain validators who are assigned to the parachain which has produced the PoV block as described in Section 2.4. Once parachain validators have validated a parachain's PoV block successfully, they have to announce that according to the procedure described in Section 2.4.1 where they generate a candidate receipt that includes the parachain header with the new state root and the XCMP message root. This candidate receipt and attestations, which carries signatures from other parachain validators is put on the relay chain.

As soon as the proposal of a PoV block is on-chain, the parachain validators break the PoV block into erasure-coded pieces as described in Section **??** and distribute them among all validators. See Section **??** for details on how this distribution takes place.

Once validators have received erasure-coded pieces for several PoV blocks for the current relay chain block (that might have been proposed a couple of blocks earlier on the relay chain), they announce that they have received the erasure coded pieces on the relay chain by voting on the received pieces, see Section 2.7 for more details.

As soon as $> 2/3$ of validators have made this announcement for any parachain block we *act on* the parachain block. Acting on parachain blocks means we update the relay chain state based on the candidate receipt and considered the parachain block to have happened on this relay chain fork.

After a certain time, if we did not collect enough signatures approving the availability of the parachain data associated with a certain candidate receipt we decide this parachain block is unavailable and allow alternative blocks to be built on its parent parachain block, see **??**.

The secondary check described in Section 2.9, is done by one or more randomly assigned validators to make sure colluding parachain validators may not get away with validating a PoV block that is invalid and not keeping it available to avoid the possibility of being punished for the attack.

During any of the phases, if any validator announces that a parachain block is invalid then all

validators obtain the parachain block and check its validity, see Section **??** for more details.

All validity and invalidity attestations go onto the relay chain, see Section 2.8 for details. If a parachain block has been checked at least by certain number of validators, the rest of the validators continue with voting on that relay chain block in the GRANDPA protocol. Note that the block might be challenged later.

2.2 Preliminaries

Definition 9 *In the remainder of this chapter we assume that ρ is a Polkadot Parachain and B is a block which has been produced by ρ and is supposed to be approved to be ρ's next block. By R_{rho} we refer to runtime code of parachain ρ as a WASM Blob.*

Definition 10 *The witness proof of block B, denoted by π_B, is the set of all the external data which has gathered while the ρ runtime executes block B. The data suffices to re-execute R_{rho} against B and achieve the final state indicated in the $H(B)$.*

This witness proof consists of light client proofs of state data that are generally Merkle proofs for the parachain state trie. We need this because validators do not have access to the parachain state, but only have the state root of it.

Definition 11 *Accordingly we define the* **proof of validity block** *or* **PoV** *block in short, PoV_B, to be the tuple:*

$$(B, \pi_B)$$

Definition 12 *The extra validation data v_B is an extra input to the validation function, i.e. additional data from the relay chain state that is needed.*

This extra validation data includes things like the previous parachain block header, likely including the previous state root. Parachain validators get this extra validation data from the current relay chain state. Note that a PoV block can be paired with different extra validation data depending on when and which relay chain fork it is included in. Future validators would need this extra validation data because since the candidate receipt was included on the relay chain the needed relay chain state may have changed.

Definition 13 *Accordingly we define the* **erasure coding blob** *or* **blob** *in short, $\bar{B}$ to be the tuple:*

$$(B, \pi_B, v_B)$$

Note that in the code the blob is referred to as "AvailableData".

2.3 Overal process

The Figure 2.3 demonstrates the overall process of assuring availability and validity in Polkadot TODO: complete the Diagram.

Restricted shell escape. PlantUML cannot be called. Start pdflatex/lualatex with -shell-escape.
@startuml
(*) –¿ "¡math¿Parachain Collator C_ρ Generates B and PoV_B¡/math¿" –¿ "¡math¿C_ρ sends PoV_B to ρ's validator V_ρ¡/math¿" –¿ "¡math¿V_ρ runs ρ's runtime on PoV¡/math¿" if "¡math¿PoV_B is valid¡/math¿" then –¿[true] if "¡math¿V_ρ have seen the CandidateReceipt for PoV_B¡/math¿" then –¿[true] Sign CandidateReceipt –¿[Ending process] (*)
else –¿ [False] "Gerenate CandiateReceipt" –¿[Ending process] (*)
endif else –¿[false] "¡math¿Broadcast message of invalidity for PoV_B¡/math¿" end if
–¿[Ending process] (*)
@enduml

Figure 2.1: Overall process to acheive availability and validity in Polkadot

2.4 Primary Validation

Primary validity checking refers to the process of parachain validators as defined in Definition **??** validating a parachain's PoV block as explained in Algorithm 3.

Algorithm 3 PrimaryValidation

Input: B, π_B, relay chain parent block $B_{relayparent}$

1: Retrieve v_B from the relay chain state at $B_{relayparent}$
2: Run Algorithm 4 using B, π_B, v_B

Algorithm 4 ValidateBlock

Input: B, π_B, v_B

1: retrieve the runtime code R_ρ that is specified by v_B from the relay chain state.
2: check that the initial state root in π_B is the one claimed in v_B
3: Execute R_ρ on B using π_B to simulate the state.
4: If the execution fails, return fail.
5: Else return success, the new header data h_B and the outgoing messages M.

2.4.1 Primary validity announcement

Validator v needs to perform Algorithm 5 to announce the result of primary validation to the Polkadot network.

In case that validation has been successful, the announcement will either be in the form of sending the candidate receipt for block B as defined in Definition 14 to the relay chain or confirm a candidate receipt sent in from another parachain validators for this block according to Algorithm 7. However, if the validation fails, v reacts by executing Algorithm 8.

Definition 14 *Candidate Receipt is a proposal for B, TBS.*

Algorithm 5 PrimaryValidationAnnouncement

Input:

1: TBS

Algorithm 6 SendPoVCandidateReceipt

Input:

1: TBS

2.4.2 Inclusion of candidate receipt on the relay chain

Definition 15 *Parachain Block Proposal, noted by P^B_{rho} is a candidate receipt for a parachain block B for a parachain ρ along with signatures for at least 2/3 of $\mathcal{V}_\rho$.*

A block producer which observe a Parachain Block Proposal as defined in definition 15 may/should include the proposal in the block they are producing according to Algorithm 9 during block production procedure.

2.4.3 Primary Validation Disagreement

Parachain validators need to keep track of candidate receipts (see Definition 14) and validation failure messages of their peers. In case, there is a disagreement among the parachain validators about $\bar{B}$, all parachain validators must invoke Algorithm 10

2.5 Availability

When a $v \in \mathcal{V}_\rho$ observes that a block containing parachain block candidate receipt is included in a relay chain block RB_p then it must invoke Algorithm 11.

Definition 16 *The* **erasure encoder/decoder** *$encode_{k,n}/decoder_{k,n}$ is defined to be the Reed-Solomon encoder defined in [**?**].*

Definition 17 *The* **set of erasure encode pieces** *of $\bar{B}$, denoted by:*

$$Er_B := (e_1, m_1), ..., (e_n, m_n)$$

is defined to be the output of the Algorithm 11.

Algorithm 7 ConfirmCandidateReceipt

Input:

1: TBS

Algorithm 8 AnnouncePrimaryValidationFailure

Input:

1: TBS

Algorithm 9 IncludeParachainProposal(P^B_{rho})

Input:

1: TBS

2.6 Distribution of Pieces

Following the computation of Er_B, v must construct the $\bar{B}$ Availability message defined in Definition 18. And distribute them to target validators designated by the Availability Networking Specification [**?**].

Definition 18 *PoV erasure piece message $M_{PoV_{\bar{B}}}(i)$ is TBS*

2.7 Announcing Availability

When validator v receives its designated piece for $\bar{B}$it needs to broadcast Availability vote message as defined in Definition19

Definition 19 *Availability vote message M^{Avail,v_i}_{PoV} TBS*

Some parachains have blocks that we need to vote on the availability of, that is decided by ¿ 2/3 of validators voting for availability. For 100 parachain and 1000 validators this will involve putting 100k items of data and processing them on-chain for every relay chain block, hence we want to use bit operations that will be very efficient. We describe next what operations the relay chain runtime uses to process these availability votes.

For each parachain, the relay chain stores the following data:
1) availability status, 2) candidate receipt, 3) candidate relay chain block number
where availability status is one of {no candidate, to be determined, unavailable, available} .
For each block, each validator v signs a message
Sign(bitfield b_v, block hash h_b)
where the ith bit of b_v is 1 if and only if

1. the availability status of the candidate receipt is "to be determined" on the relay chain at block hash h_b **and**
2. v has the erasure coded piece of the corresponding parachain block to this candidate receipt.

These signatures go into a relay chain block.

Algorithm 10 PrimaryValidationDisagreement

Input:

1: TBS

Algorithm 11 ERASURE-ENCODE($\bar{B}$, n

Input: $\bar{B}$: blob defined in Definition 13

1: TBS

2.7.1 Processing on-chain availability data

This section explains how the availability attestations stored on the relay chain, as described in Section ??, are processed as follows:

Algorithm 12 Relay chain's signature processing

1: The relay chain stores the last vote from each validator on chain. For each new signature, the relay chain checks if it is for a block in this chain later than the last vote stored from this validator. If it is the relay chain updates the stored vote and updates the bitfield b_v and block number of the vote.

2: For each block within the last t blocks where t is some timeout period, the relay chain computes a bitmask bm_n (n is block number). This bitmask is a bitfield that represents whether the candidate considered in that block is still relevant. That is the ith bit of bm_n is 1 if and only if for the ith parachain, (a) the availability status is to be determined and (b) candidate block number $\leq n$

3: The relay chain initialises a vector of counts with one entry for each parachain to zero. After executing the following algorithm it ends up with a vector of counts of the number of validators who think the latest candidates is available.

   1. The relay chain computes b_v and bm_n where n is the block number of the validator's last vote
   2. For each bit in b_v and bm_n
      - add the ith bit to the ith count.

4: For each count that is $> 2/3$ of the number of validators, the relay chain sets the candidates status to "available". Otherwise, if the candidate is at least t blocks old, then it sets its status to "unavailable".

5: The relay chain acts on available candidates and discards unavailable ones, and then clears the record, setting the availability status to "no candidate". Then the relay chain accepts new candidate receipts for parachains that have "no candidate: status and once any such new candidate receipts is included on the relay chain it sets their availability status as "to be determined".

Based on the result of Algorithm **??** the validator node should mark a parachain block as either available or eventually unavailable according to definitions 20 and **??**

Definition 20 *Parachain blocks blocks for which the corresponding blob is noted on the relay chain to be available, meaning that the candidate receipt has been voted to be available by 2/3 validators.*

After a certain time-out in blocks since we first put the candidate receipt on the relay chain if there is not enough votes of availability the relay chain logic decides that a parachain block is unavailable, see 12.

Definition 21 *An unavailabile parachain block is TBS*

/syedSo to be clear we are not announcing unavailability we just keep it for grand pa vote

2.8 Publishing Attestations

We have two type of attestations, primary and secondary. Primary attestations are signed by the parachain validators and secondary attestations are signed by secondary checkers and include the VRF that assigned them as a secondary checker into the attestation. Both types of attestations are included in the relay chain block as a transaction. For each parachain block candidate the relay chain keeps track of which validators have attested to its validity or invalidity.

2.9 Secondary Approval checking

Once a parachain block is acted on we carry the secondary validity/availability checks as follows. A scheme assigns every validator to one or more PoV blocks to check its validity, see Section 2.9.3 for details. An assigned validator acquires the PoV block (see Section **??**) and checks its validity by comparing it to the candidate receipt. If validators notices that an equivocation has happened an additional validity/availability assignments will be made that is described in Section2.9.5.

2.9.1 Approval Checker Assignment

Validators assign themselves to parachain block proposals as defined in Definition 15. The assignment needs to be random. Validators use their own VRF to sign the VRF output from the current relay chain block as described in Section 2.9.2. Each validator uses the output of the VRF to decide the block(s) they are revalidating as a secondary checker. See Section **??** for the detail.

In addition to this assignment some extra validators are assigned to every PoV block which is descirbed in Section **??**.

2.9.2 VRF computation

Every validator needs to run Algorithm 13 for every Parachain ρ to determines assignments. TODO: Fix this. It is incorrect so far.

Algorithm 13 VRF-FOR-APPROVAL(B, z, s_k)

Input: B: the block to be approved
z: randomness for approval assignment
s_k: session secret key of validator planning to participate in approval

1: $(\pi, d) \leftarrow VRF(H_h(B), sk(z))$
2: **return** (π, d)

Where VRF function is defined in [**?**].

2.9.3 One-Shot Approval Checker Assignemnt

Every validator v takes the output of this VRF computed by 13 mod the number of parachain blocks that we were decided to be available in this relay chain block according to Definition 20 and executed. This will give them the index of the PoV block they are assigned to and need to check. The procedure is formalised in 14.

Algorithm 14 OneShotAssignment

Input:

1: TBS

2.9.4 Extra Approval Checker Assigment

Now for each parachain block, let us assume we want $\#VCheck$ validators to check every PoV block during the secondary checking. Note that $\#VCheck$ is not a fixed number but depends on reports from collators or fishermen. Lets us $\#VDefault$ be the minimum number of validator we want to check the block, which should be the number of parachain validators plus some constant like 2. We set

$$\#VCheck = \#VDefault + c_f * \text{total fishermen stake}$$

where c_f is some factor we use to weight fishermen reports. Reports from fishermen about this

Now each validator computes for each PoV block a VRF with the input being the relay chain block VRF concatenated with the parachain index.

For every PoV bock, every validator compares $\#VCheck - \#VDefault$ to the output of this VRF and if the VRF output is small enough than the validator checks this PoV blocks immediately otherwise depending on their difference waits for some time and only perform a check if it has not seen $\#VCheck$ checks from validators who either 1) parachain validators of this PoV block 2) or assigned during the assignment procedure or 3) had a smaller VRF output than us during this time.

More fisherman reports can increase $\#VCheck$ and require new checks. We should carry on doing secondary checks for the entire fishing period if more are required. A validator need to keep track of which blocks have $\#VCheck$ smaller than the number of higher priority checks performed. A new report can make us check straight away, no matter the number of current checks, or mean that we need to put this block back into this set. If we later decide to prune some of this data, such as who has checked the block, then we'll need a new approach here.

Algorithm 15 OneShotAssignment

Input:

1: TBS

2.9.5 Additional Checking in Case of Equivocation

In the case of a relay chain equivocation, i.e. a validator produces two blocks with the same VRF, we do not want the secondary checkers for the second block to be predictable. To this end we use the block hash as well as the VRF as input for secondary checkers VRF. So each secondary checker

is going to produce twice as many VRFs for each relay chain block that was equivocated. If either of these VRFs is small enough then the validator is assigned to perform a secondary check on the PoV block. The process is formalized in Algorithm 16

Algorithm 16 EquivocatedAssignment

Input:

1: TBS

2.10 The Approval Check

Once a validator has a VRF which tells them to check a block, they announce this VRF and attempt to obtain the block. It is unclear yet whether this is best done by requesting the PoV block from parachain validators or by announcing that they want erasure coded pieces.

Retrieval

There are two fundamental ways to retrieve a parachain block for checking validity. One is to request the whole block from any validator who has attested to its validity or invalidity. Assigned appoval checker v sends RequestWholeBlock message specified in Definition **??** to parachain validator in order to receive the specific parachain block. Any parachain validator receiving must reply with PoVBlockRespose message defined in Definition 22

Definition 22 *PoV Block Respose Message TBS*

The second method is to retrieve enough erasure coded pieces to reconstruct the block from them. In the latter cases an announcement of the form specified in Definition has to be gossiped to all validators indicating that one needs the erasure coded pieces.

Definition 23 *Erasure coded pieces request message TBS*

On their part, when a validator receive a erasure coded pieces request message it response with the message specified in Definition 24.

Definition 24 *Erasure coded pieces response message TBS*

Assigned appoval checker v must retrieve enough erasure pieces of the block they are verifying to be able to reconstruct the block and the erasure pieces tree.

Reconstruction

After receiving $2f + 1$ of erasure pieces every assigned approval checker v needs to recreate the entirety of the erasure code, hence every v will run Algorithm **??** to make sure that the code is complete and the subsequently recover the original $\bar{B}$.

Algorithm 17 RECONSTRUCT-POV-ERASURE(S_{Er_B})

Input: $S_{Er_B} := (e_{j_1}, m_{j_1}), \cdot, (e_{j_k}, m_{j_k}))$ such that $k > 2f$

1: $\bar{B} \rightarrow$ ERASURE-DECODER$(e_{j_1}, \cdots, e_{j_k})$
2: **if** ERASURE-DECODER **failed then**
3: ANNOUNCE-FAILURE
4: **return**
5: **end if**
6: $Er_B \rightarrow$ ERASURE-ENCODER$(\bar{B})$
7: **if** VERIFY-MERKLE-PROOF(S_{Er_B}, Er_B) **failed then**
8: ANNOUNCE-FAILURE
9: **return**
10: **end if**
11: **return** $\bar{B}$

2.10.1 Verification

Once the parachain bock has been obtained or reconstructed the secondary checker needs to execute the PoV block. We declare a the candidate receipt as invalid if one one the following three conditions hold: 1) While reconstructing if the erasure code does not have the claimed Merkle root, 2) the validation function says that the PoV block is invalid, or 3) the result of executing the block is inconsistent with the candidate receipt on the relay chain.

The procedure is formalized in Algorithm

Algorithm 18 REVALIDATINGRECONSTRUCTEDPOV

Input:

1: TBS

If everything checks out correctly, we declare the block is valid. This means gossiping an attestation, including a reference that identifies candidate receipt and our VRF as specified in Definition 25.

Definition 25 *Secondary approval attetstion message TBS*

2.10.2 Process validity and invalidity messages

When a Black produced receive a Secondary approval attetstion message, it execute Algorithm 19 to verify the VRF and may need to judge when enough time has passed.

Algorithm 19 VERIFYAPPROVALATTESTATION

Input:

1: TBS

These attestations are included in the relay chain as a transaction specified in

Definition 26 *Approval Attestation Transaction TBS*

Collators reports of unavailability and invalidty specified in Definition TODO: Define these messages also go onto the relay chain as well in the format specified in Definition

Definition 27 *Collator Invalidity Transaction TBS*

Definition 28 *Collator unavailability Transaction*

2.10.3 Invalidity Escalation

When for any candidate receipt, there are attestations for both its validity and invalidity, then all validators acquire and validate the blob, irrespective of the assignments from section by executing Algorithm **??** and 18.

We do not vote in GRANDPA for a chain were the candidate receipt is executed until its vote is resolved. If we have n validators, we wait for $> 2n/3$ of them to attest to the blob and then the outcome of this vote is one of the following:

If $> n/3$ validators attest to the validity of the blob and $\leq n/3$ attest to its invalidity, then we can vote on the chain in GRANDPA again and slash validators who attested to its invalidity.

If $> n/3$ validators attest to the invalidity of the blob and $\leq n/3$ attest to its validity, then we consider the blob as invalid. If the rely chain block where the corresponding candidate receipt was executed was not finalised, then we never vote on it or build on it. We slash the validators who attested to its validity.

If $> n/3$ validators attest to the validity of the blob and $> n/3$ attest to its invalidity then we consider the blob to be invalid as above but we do not slash validators who attest either way. We want to leave a reasonable length of time in the first two cases to slash anyone to see if this happens.