Proof-of-Loss

A Purely Symbolic
Block-Chaining Algorithm
for Monetary Consensus

Mirelo Deugh Ausgam Valis

Introduction

What is proof-of-loss?

It is a consensus algorithm based on a chain of transaction blocks, like Bitcoin or Peercoin. However, it fundamentally differs from either currency, for using *lost spending rights* to both reward block chaining and determine block-chaining odds.

In Bitcoin, each miner sells the conclusion (first confirmation) of transactions for fees. However, those fees could also be buying the right to that conclusion during its occurrence. This valid interpretation is optional since, in Bitcoin, the right to transact is always created and destroyed in the same block.

Indeed, although transaction rights are the reward for chaining each block, if only saleable in the same block earning them, they become redundant, hence irrelevant.

In proof-of-loss, on the contrary, transaction rights are never created and destroyed in the same block:

- They are only saleable in their creating block's descendants.
- They must sell before any pending transactions can conclude.

Then, for already existing despite not yet saleable, those rights are no longer irrelevant.

But what is the right to transact, and how can it sell? If selling requires pricing, which requires valuing, which requires measuring, then how is this right measured?

In proof-of-loss, transaction rights are merely the size in bytes of each transaction allowed to conclude:

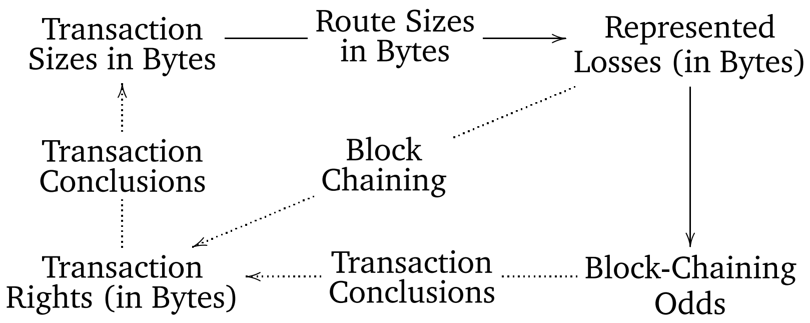
- If that conclusion did not yet happen, then those rights remain.
- If it already did, then they have become their loss.

However, this loss is not merely the size in bytes of each concluded transaction, but rather that size divided among all spendable outputs from this transaction proportionally to their sizes in bytes. Then:

- For being a representation of the same loss, each of those outputs becomes a “**r**ights **out**put **e**xtinction,” or a *route*.
- As later specified (see section 3 on page 5), the odds of chaining each block will depend not on the balance of those outputs as in proof-of-stake, but rather on the loss they represent in bytes.
- As also later specified (see section 4 on page 7), the reward for chaining each block will be the right to make transactions of a total size combining:
  - ◇ The loss represented in that block by all spendable outputs from paid transactions.
  - ◇ The loss represented by the route chaining the same block.

Thus, proof-of-loss is a *closed* or *self-referential* algorithm, in which only lost rights can (as themselves) reward block chaining or (as their loss) determine block-chaining odds. Even so, transaction volume in bytes can still grow, since the protocol restores those rights both when lost and each time their loss chains a block.

So the lifecycle of all spending rights is as follows (the solid lines mean synchrony, and the dotted ones, asynchrony):



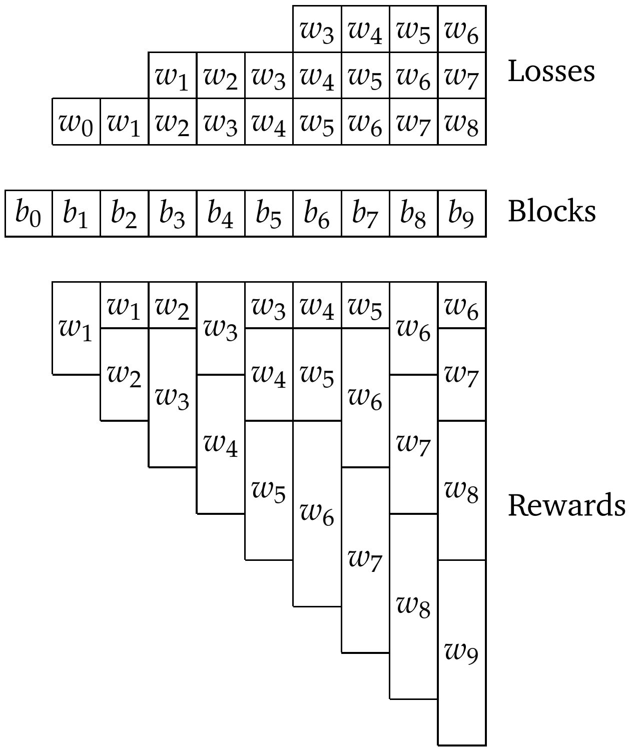
Likewise, the fraction of each previous reward W for sale in the current block depends exclusively on W and its earning route r, by always having the smaller size in bytes between:

- The rights not yet surrendered from W.
- The loss represented by r.

For example, as later confirmed (see section 3 on page 4), if r is the only route of a transaction t, then r's represented loss is t's total size in bytes. Thus:

- Let all transactions have the same size in bytes and a single route.
- Let the first block have a single transaction.

Then, the maximum transaction volume in bytes of the first nine blocks is as follows (with rewards numbered after their source block and losses named after their partly consumed reward):



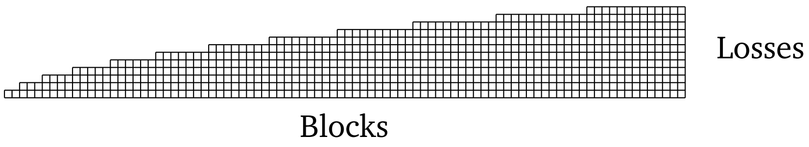
Where:

1. The route r_0 chaining the first block b_1 earns the reward w_1, of which the size in bytes restores the rights consumed by the single transaction t_1 (spending r_0) in b_1 plus those having their loss l_0 (at r_0's funding transaction t_0 in the dummy block b_0) represented by r_0:

- Again as later specified (see section 8 on page 14), t_1's input instructs the system to redistribute r_0's collected fees to t_1's route r_1.
- As later confirmed (see section 4 on page 7), w_1's fraction for sale in the second block b_2 has the size in bytes of l_0, so b_2 has a single transaction t_2 of the size in bytes of t_1.
- Since no longer spendable, r_0 cannot keep representing l_0, nor hence chain any blocks after b_1.

2. Then, t_1's route r_1 chains b_2, thus earning the second reward w_2, of which the size in bytes restores w_1's fraction consumed by the single transaction t_2 (spending r_1) in b_2 plus the loss l_1 (at t_1) represented by r_1:
   - t_2's input instructs the system to redistribute r_1's collected fees to t_2's route r_2.
   - The third block b_3 puts for sale equal parts of w_1 and w_2.
3. For chaining b_3, r_2 earns the third reward w_3, of which the size in bytes restores the rights partly consumed from w_1 and w_2 by the now two transactions t_3 and t_4 in b_3 plus the loss l_2 (at t_2) represented by r_2:
   - t_3 spends r_2, then t_4 funds its route r_4 with t_3's one r_3.
   - t_3's input instructs the system to redistribute r_2's collected fees to r_3, then t_4's one, to redistribute them to r_4.
4. For chaining the fourth block b_4, r_4 earns the fourth reward w_4, of which the size in bytes restores the rights partly consumed from w_2 and w_3 by again two transactions t_5 and t_6 in b_4 plus the loss l_4 (at t_4) represented by r_4:
   - t_5 spends r_4, then t_6 funds its route r_6 with t_5's one r_5.
   - t_5's input instructs the system to redistribute r_4's collected fees to r_5, then t_6's one, to redistribute them to r_6.
5. And so on.

This way, the maximum transaction volume in bytes of the first 90 blocks is as follows:



Despite seemingly unrealistic, this example perfectly describes a currency bootstrap scenario. After the bootstrap, transaction volume in bytes can grow faster or slower or even shrink since:

- A block can contain any number of paid transactions, each with a different size in bytes and number of routes or inputs.
- As later specified, rewards failing to sell are partly revocable (see section 4 on page 8) and inheritable (still section 4 on page 8).
- As also later specified (see section 7 on page 13), people can prevent the currently saleable fraction of any — overpriced — rewards from selling in their chained blocks.

The use of transaction rights makes proof-of-loss *purely symbolic*, meaning the algorithm need not rely on losses external to the block chain, as proof-of-stake and -work do, but merely on the symbols representing those losses. Instead of relying on amassed stake or hashing power to (unprovenly) represent lost wealth, proof-of-loss relies on the size in bytes of concluded transactions to (provenly) represent lost spending rights.

This purely symbolic design logically evolves into systemic policies, like *inactivity fees* (on page 23), *intrinsic checkpoints* (on page 26), and an *adaptive monetary policy* (on page 27). But not before also naturally addressing the problems:

- Of a lacking organic block size limit, by making transaction volume in bytes both economically prioritized (see page 11) and dependent on its past.
- Of "nothing at stake," by making the child of a block provenly chained in parallel inherit both all earned rights and seized fees in its parent (see page 19).

1 Block-Chaining Incentives

The coinbase parameter of Bitcoin's genesis block reads:

> The Times 03/Jan/2009 Chancellor on brink of second bailout for banks

This reference indicates Bitcoin was designed to at least partly prevent a future global monetary crisis like that of 2008. However, any such crisis is only the ultimate result of money falsely becoming its represented wealth,[1] hence of a confusion Bitcoin could never entirely avoid as its proof-of-work incentive model:

1. Makes hashing-power create or collect its monetary reward, of which it cannot produce the represented wealth.
2. Requires those earnings to be worth more than paid for that power, so by self-expanding, money becomes indistinguishable from the wealth thus additionally represented.

Eventually, by eliminating hashing-power mediation, proof-of-stake would reduce this design problem to its essence. For example, in Peercoin,[2] the reward for chaining each block depends directly on the size and age of the stake enabling that chaining. This model can appear to be fair for keeping all its rewards invariable in relative size, hence by making them purely relative. However, these rewards could only have such an irrelevant absolute size by being market prices, which must rather be variable also in relative size. Thus, if still incompletely variable, the same rewards must eventually cause recursive increases in wealth inequality.

For example, Peercoin tends to reward people with a newly created or *minted* 1% of their block-chaining stake per year. So:

- Let Peercoin be the only form of money for both Bob and Alice.

[1] This fact would not prevent some people from likening Bitcoin's monetary system to a "decentralized autonomous *company*" (DAC) as if money could indeed produce its represented wealth.

[2] Also known as Peer-to-Peer Coin or PPCoin, both sharing the acronym PPC, Peercoin was the first block-chained monetary system to use proof-of-stake.

- Let Bob's income be 100 monetary units or *coins* per month while his expenses are 80% of his income.
- Let Alice's income be 400 coins per month while her expenses are 50% of her income.
- For simplicity, neither let Bob nor Alice have any savings — which Alice is more likely to have.

Then, Bob and Alice will be able to stake 20 and 200 coins per month, respectively, so most likely:

1. Alice's minted reward will exceed Bob's by 900%, even though her income exceeds his by only 300%.
2. In addition to earning an undue proportion of their combined mintage, Alice will be able to stake this excess reward in the future, thus increasing it exponentially.

The only way to prevent the resulting price inflation from causing an increasingly uneconomical wealth transfer from Bob to Alice is letting a market decide the block-chaining reward, so proof-of-stake stops raising the following questions:

1. How much newly created money must people be able to earn as their incentive to engage in block chaining? Is 1% of stake per year enough? If not, then what about 6%?
2. How can the chosen percentage neither underestimate people's greed nor overestimate their willingness to fund uneconomical, undue rewards through the resulting price inflation?

Still, unless each block-chaining reward consists exclusively of transaction fees,[3] no market could remain its only determinant since prices are everything a market can determine completely. Hence, only markets for the product costing those fees can economically decide that reward, yet what could people exchange in these markets? What have transaction fees ever priced?

[3] In Bitcoin, although transaction fees were always part of the block-chaining reward, they will not be all of it until mining yields become less than the current coin precision, by halving each four years from 50 coins in 2009.

2 Transaction Rights

The essential purpose of chaining blocks is neither to create nor even to earn additional money, but only to conclude (first-confirm) transactions. However, to optimize monetary consensus, people must always have an incentive to engage in block chaining, even while having no need to transact. Hence, as concluding their unowned transactions cannot reward them:

1. These people must have an incentive to chain blocks other than satisfying anyone's need for monetary transfers.
2. To never let extrinsic reasons make it fail, this additional incentive must remain as essential a purpose to block chaining as concluding those transfers.

Indeed, all spenders whose transactions depend on block chaining have already an *impermanent* incentive to chain blocks. Then, how to make that incentive *permanent* to each one of them? Only by requiring each transaction conclusion to be *sold*, so those offering it can earn at least what they would pay for a similar offer.

Still, the mere possibility of a transaction conclusion is never saleable. Thus, if that conclusion must sell before it can occur, then how could it sell? Only as a *right*: the right to the same conclusion, or to transact.[4] So people chaining blocks must always be rewarded with transaction rights instead of transaction fees, and only later sell those rights for those fees.

[4] Except as a right, each transaction conclusion must become indistinguishable from its product, since only saleable as a concluded transaction — as in Bitcoin — rather than as merely the right to transact.

3 Proof-of-Loss

However, despite transaction rights being the only block-chaining reward left, amassing stake or hashing-power does not require selling those rights. Instead, it can result from selling actual wealth — that with purely material utilities. Hence, if amassed stake or hashing-power can still determine block-chaining rewards, even by only affecting block-chaining odds, all money thus earned remains indistinguishable from what it can buy.

For that money to ever be distinguishable from its represented wealth, block-chaining odds must depend instead on used hence lost transaction rights, which are then recoverable either by chaining blocks or paying others for doing so, regardless of amassed stake or hashing-power. Indeed, money cannot remain decentralized unless each of its users either keeps it spendable or pays others for doing so, regardless of this user's accumulated wealth.

While conversely, since transaction rights (hence block chaining) depend on money being spendable, any lost such rights (hence the resulting block-chaining odds) can only take the form of spendable outputs, each of which is then a *route*, meaning a "**r**ights **out**put **e**xtinction" r, as follows:

$$l_r = z_t \times \frac{z_r}{Z_r}$$

Where:

- l_r means r's represented loss in bytes.
- z_t means the total size in bytes of r's transaction t.
- z_r means r's total size in bytes.
- Z_r means the combined size in bytes of all of t's routes, whether spent or not — including r.

Indeed, only the size in bytes of each concluded transaction is a *purely symbolic* block-chaining resource, by representing nothing external to the public chain of blocks, including any wealth — which must remain both external to that chain and optionally private.

Still, this protocol must always:

- Minimize the block-chaining odds of each recently rewarded or concluded route *r*, so its owners (who can have concluded it) are unlikely to monopolize the block-chaining process.
- Reduce these odds proportionally to the rights not surrendered from those earned in *r*'s most recently chained or else concluding block, so *r*'s owners (who can have concluded it) are unlikely to monopolize the existing transaction rights.

Additionally, the same protocol must always:

1. Reduce *r*'s block-chaining odds proportionally to its block depth, so people are unlikely to monopolize future block chaining by accumulating spendable outputs.
2. Increase these odds proportionally to the already surrendered rights from those earned in *r*'s most recently chained or else concluding block, so any depthless routes consuming those rights are unlikely to monopolize the block-chaining process.

Then, *r*'s block-chaining odds must be:

$$x_r = x_g \times l_r \times \left(\frac{l_r}{W_r} + \left(\left(1 - \frac{l_r}{W_r}\right) \times \frac{W_S}{W_r}\right)\right) \times \frac{\frac{W_S}{l_r} + 1}{d_r}$$

Where:

- x_r means *r*'s difficulty target, which is inversely proportional to *r*'s block-chaining difficulty.
- x_g means the general difficulty target, which must self-readjust to maintain a constant block interval.
- l_r means *r*'s represented loss in bytes (as already specified on the preceding page).
- W_r means the total earned rights in *r*'s highest chained or else concluding block.
- W_S means the total surrendered rights from W_r.

- d_r means r's block depth, which is always the depth of r's concluding rather than highest chained block.[5]

This use of transaction rights as the only block-chaining reward and of their loss as the only factor of block-chaining odds utterly defines the proof-of-loss protocol. Further specifying this protocol must never make that definition invalid.

So proof-of-loss alone can distinguish any block-chained money from its represented wealth, for being the only block-chaining monetary consensus algorithm ever to prove the loss of something purely symbolic, which is everything always distinguishable from any wealth. For the same reason, although every algorithm for block-chained monetary consensus must assume a proof of loss, only proof-of-loss can provide it.[6] Indeed:

1. No other such algorithm can directly represent what it must prove was lost — nor then be purely symbolic. For example, amassed stake or hashing-power could never be the lost wealth that respectively proof-of-stake or -work must prove.

2. Every loss represented by something other than itself must be uncertain — since not purely symbolic. For example, in proof-of-stake or -work, the costs respectively of amassing stake or hashing-power can always be overvalued.

However, nothing could be uncertain to whom it was proven. So, in a block-chained monetary system, lost spending rights (represented by the size in bytes of each concluded transaction divided among its routes proportionally to their sizes in bytes) are the only possible form of *proof-of-loss*.

[5] Unlike a block-chaining reward in money, one in rights need not (nor otherwise could) add to its earning balance by transferring it.

[6] Knowingly or not, all other block-chaining consensus algorithms, regardless of their proof of choice, intend to be forms of proof-of-loss. However, they could never *prove* any loss. Instead, they can only prove something that *indicates* one.

4 Block-Chaining Rewards

To prevent money from falsely becoming its represented wealth, not only the odds of chaining a block but also the size in bytes of the resulting reward must depend entirely on lost rights. Otherwise, these rights could become that reward even without being lost, thus letting the money buying their loss create more of them — so their selling would make that money again self-expand. Hence, the same reward can only be the paid rights:

- Of which the loss enables the block chaining thus rewarded.
- Lost on all transactions buying rights in the chained block.[7]

Additionally, as either possibility alone would prevent transaction volume in bytes from ever increasing, the reward for chaining each block must always combine:

- The paid part of that volume concluded in this block.
- The size in bytes of the loss chaining the same block.

Still, a route must not be able to sell its earned rights in its chained blocks, or the resulting losses would become its additional rights, which could recursively become ever more such losses. Likewise, if any rights currently for sale exceeded their earning loss, then their market share would not be proportional to that loss, nor hence the resulting price competition to block-chaining decentralization. Thus, each reward's fraction for sale in a subsequent block must always be the smaller in bytes between that reward's rights not yet surrendered and the loss their earning route represents.

However, this policy must also:

- Prevent total existing rights from increasing unnecessarily, which could make their price fail as a block-chaining incentive.
- Allow them to decrease at most by their possible increase, to avoid unduly prioritizing either variation.

[7] So the size in bytes of all block-chaining rewards must exclude that of any transactions originated by the system.

So, since each block-chaining reward W can later increase total sold rights by at most its earning loss l, the rights from W that fail to sell must be revoked at most by $l \times 2$, as follows:

$$K_c = \begin{cases} 0 & \text{if } c \leqslant u \\ S_c - B_c & \text{if } c > u \text{ and } S_c - B_c \leqslant 2l - \sum K_{h\,<\,c} \\ 2l - \sum K_{h\,<\,c} & \text{if } c > u \text{ and } S_c - B_c > 2l - \sum K_{h\,<\,c} \end{cases}$$

Where:

- K_c means W's rights revoked at the current block height c.
- u means l's chained block's height, at which W is unsaleable.[8]
- S_c ($\leqslant l$) means W's fraction for sale at c.
- B_c means S_c's rights bought at c.
- $K_{h\,<\,c}$ means W's rights revoked at each block height h lower than c. So, for $c = u + 3$:

$$\sum K_{h\,<\,c} = K_{u\,+\,1} + K_{u\,+\,2}$$

Finally, to minimize the selling opportunities of any overpriced rights while maximizing their opportunities to sell at a lower price, the reward for chaining a block b must inherit any unsold rights failing to sell in b's parent p but not revoked, as follows:

$$U_p = S_p - \left(B_p + K_p\right)$$

Where:

- U_p means the unsold rights inherited by b's chaining reward from p's one.
- S_p means the rights for sale in p, whether inherited or not by the reward for chaining b or p from that for chaining their parents.
- B_p means S_p's fraction bought in p.
- K_p means S_p's fraction revoked in p.

[8] As already specified on the previous page, all transaction rights must be unsaleable in the same block of which they reward the chaining.

5 Price Negotiation

In Bitcoin, for not being formally asked, the price miners charge for transaction rights must remain implicit in paid transaction fees, which hence:

- Can only be an estimate of this informally charged price.
- Can only be an implicit bid since one on informal asks.

While conversely:

- When spenders overvalue a transaction conclusion — as they are more likely to do the less they can wait for it — miners cannot price it lower. Then, transaction costs uneconomically rise.
- When spenders undervalue that conclusion — as they are more likely to do the more they can wait for it — miners cannot price it higher. Then, transaction costs uneconomically fall.

So, for the same conclusion always to be priced economically:

- Selling it requires formalizing its asked price, by publicizing *transaction asks*.
- Buying it requires formalizing its bid price, by converting every paid transaction fee from an implicit ask estimate into an explicit ask payment.

Indeed, only this way people must always negotiate the right to transact in the market, since:

- To maximize their gains, they must make no asks above a maximum bid.
- To minimize their wait for transaction conclusions, they must make no bids below a minimum ask.

6 Block Chaining

So each paid transaction must inform not the asked fees it pays but rather its maximum payable ones, or its formal bid on its price. Then, for each block b to contain a valid combination of bids and asks, its total asked fees need only not exceed its total bid ones. Hence, b's every ask k must collect in b the following fees:

$$f_b = F_b \times \frac{f_z}{F_z}$$

Where:

- f_b means k's total earned transaction fees in b.
- F_b means the total bid such fees in b, as follows:

$$F_b = \left(f_{t_1} \times z_{t_1}\right) + \left(f_{t_2} \times z_{t_2}\right) + \cdots + \left(f_{t_n} \times z_{t_n}\right)$$

  Where:

  ◇ $f_{t_1}, f_{t_2}, \ldots, f_{t_n}$ mean the maximum payable fee per byte for each of b's concluded transactions $t_1, t_2, \ldots, t_n$.

  ◇ $z_{t_1}, z_{t_2}, \ldots, z_{t_n}$ mean the total sizes in bytes of $t_1, t_2, \ldots, t_n$.

- f_z means k's total charged fees in b, as follows:

$$f_z = f_k \times z_b$$

  Where:

  ◇ f_k means k's charged fee per byte of transaction rights.

  ◇ z_b means the size of k's total sold rights in b, as follows:

$$z_b = \frac{z_k}{Z_k} \times Z_t$$

  Where:

- z_k means the total size of k's rights for sale in b.
- Z_k means the size of the combined rights for sale in b by all of b's asks — including k.
- Z_t ($\leqslant Z_k$) means the combined size in bytes of all transactions buying rights in b.

- Fz ($\leqslant F_b$) means the total fees charged in b, as follows:

$$Fz = fz_1 + fz_2 + \cdots + fz_n$$

Where $fz_1, fz_2, \ldots, fz_n$ mean the same as fz on the preceding page but each calculated for one of b's asks $k_1, k_2, \ldots, k_n$ — again including k.

However:

1. Older transaction inputs must have priority over newer ones, or a spend could wait endlessly for its conclusion.
2. Smaller independently concluded losses must take precedence over larger ones, or it would still be possible to monopolize block chaining by displacing other people's losses with corresponding ones represented in fewer transactions by spendable outputs belonging to the monopolists.

Thus, to provide a combined incentive to these two required forms of transaction prioritization, b's chaining route r must seize the following fees from F_b:

$$fr = \begin{cases} \tilde{f}_b \times Q_b & \text{if } \tilde{f}_b \times Q_b \leqslant F_b \\ F_b & \text{otherwise} \end{cases}$$

Where:

- fr means r's collected fees in b.
- $\tilde{f}_b$ means the *median* f_b.[9]

[9] A median is the center of an ordered list k of possibly repeated values. If the number of those values is odd, then their median is the single value at the center of k. Otherwise, it is the average of k's two intermediate values. For example:
- With $k = \{1, 1, 1, 3, 4\}$, the median of its values is 1.
- With $k = \{1, 1, 1, 3, 4, 200\}$, the median of its values is $(1 + 3) \div 2 = 2$.

- Q_b means the prioritization quotient, as follows:

$$Q_b = \frac{\overline{d}_i}{\tilde{d}_i} \times \frac{\tilde{z}_t}{\overline{z}_t}$$

  Where:

  - ⋄ $\overline{d}_i$ means the *average* block depth of all inputs to transactions buying rights in b.[10]
  - ⋄ $\tilde{d}_i$ means the median such depth.
  - ⋄ $\tilde{z}_t$ means the median size in bytes of those transactions.
  - ⋄ $\overline{z}_t$ means the average size in bytes of the same transactions.

This algorithm provides the following benefits:

- By requiring bids and asks in each block to collectively rather than individually combine:
  - ⋄ It maximizes transaction volume in bytes, for letting some of those bids or asks conclude respectively below and above any of their asked or bid prices.
  - ⋄ It maximizes the downward pressure on fees, for making higher bids pay for lower ones.
- It enforces transaction prioritization with economic incentives, which are then intrinsic rather than (as in Bitcoin) extrinsic to the protocol.

[10] The average of any values is their sum divided by their number. For example, if an ordered list k consists of possibly repeated values, then:
- With $k = \{1, 1, 1, 3, 4\}$, the average of its values is $10 \div 5 = 2$.
- With $k = \{1, 1, 1, 3, 4, 200\}$, the average of its values is $210 \div 6 = 35$.

7 Transaction Asks

Each transaction ask must be a section of the block in which it earned its priced rights. Otherwise, those rights would tend to be overpriced, as they could be repriced lower if necessary. Still:

- Intentionally or not, people can always overprice their earned rights, even without being able to reprice them lower.
- A falling transaction volume in bytes can also make the same rights eventually overpriced.

Then, some transaction rights can become unsaleable. So, to avoid still requiring any blocks to sell them, every block must inform a (possibly empty) list of block depths corresponding to its excluded transaction asks. Otherwise, those unsaleable rights could:

- Stop the system.
- Slow block chaining, thus increasing the block interval.
- Weaken the competition between transaction-right sellers, thus reducing its lowering pressure on fees.

However, why not just prevent enough overpriced rights from selling, instead of excluding their asks? Because, as a fractional bid cannot affect which asks its block includes, neither could a fraction of an ask's rights for sale in that block do so, or fees would tend to rise.

Finally, as the resulting absence of any asks only makes their saleable rights formally overpriced, any rights then prevented from selling must still follow the same rules both of revocation (on page 8) and inheritance (on page 8) as if rather for sale.

8 Implicit Transactions

Then, as the source of all fees paid to a rewarded route in each block is any of this block's bids, being thus indeterminate unless the same block has a single bid or ask:

1. This payment must be an implicit transaction transferring those fees to that route.
2. The total output balance of each paid transaction must equal its total input balance less its informed bid.

While conversely, spending a rewarded route before it surrendered all its earned rights must have:

- All its future collected fees implicitly redistributed as instructed by the transaction input spending it, to prevent the unduly destruction of money.
- All its currently collected fees also implicitly redistributed as thus instructed, to prevent any of those fees from paying it with part of its balance.

9 Block Asks

As merely finding a proof of loss requires no block chaining, but rather just route ownership, after doing so people can immediately broadcast a *block ask* informing:

1. The primary proof-of-loss data l capable of becoming that proof by having its difficulty-matching hash signed,[11] containing the following items:
   - 1.1. The unique identifier of a route r.
   - 1.2. The present moment,[12] so each moment requires hashing another proof of loss.
   - 1.3. The current block height, so blocks at different heights cannot contain the same proof of loss.
   - 1.4. The current difficulty target for the route with the oldest not surrendered rights. The only function of this item is to minimize block re- and pre-chaining odds, by providing each proof of loss with a strongly correlated context (which results from all dependencies already specified on pages 5 and 8).[13]
2. A hash of l not above r's difficulty target, to minimize the odds of multiple routes chaining blocks at the same block height.
3. A signature of this hash by the private key to r's address, to authenticate r's proof of loss.

Then, this block ask's broadcasters can also publicize their network location as a bids verifiable destination d, so:

- Each unconfirmed transaction can be sent directly to d rather than unnecessarily broadcast.
- These people can later publicize another such *location ask*, as needed if:
  - ◇ They relocate logically, whether also physically or not.

[11] Such as a Peercoin *kernel* can prove a stake — thus indicating lost wealth — by having its difficulty-matching hash signed.
[12] Possibly the current second as in Peercoin.
[13] This context has the same function as Peercoin's less organic *stake modifier*.

 ◇ They broadcast another block ask.

Indeed, to function as a bids verifiable destination, each advertised location ask must inform:

1. A list of network (possibly IP) addresses, all of which can belong to the advertiser's peers who know its location.
2. The proof-of-loss hash L identifying a block ask b.
3. The serial number of this reference to b's proof of loss, to let people determine its current validity.
4. A difficulty-free hash N of this location ask c.
5. A signature of N by the same private key signing L in b, to authenticate c.

This algorithm provides the following benefits:

- To minimize memory consumption and even the complexity of selecting transaction destinations, people can discard any block asks informing a present moment not later than the current block's creation time, as also the corresponding location asks.
- To prevent memory overflow attacks despite always allowing alternative destinations to propagate, people receiving different location asks for the same proof of loss can relay that with the greatest unrepeated serial number, then discard the others.
- To counter denial of service attacks on the network addresses informed by current location asks, people can still:
  - ◇ Broadcast each unconfirmed transaction otherwise destined only to those addresses (the same ones possibly locating neither their advertiser nor part of its peers).
  - ◇ Use any addresses not being attacked to receive other people's broadcast candidate spends.

10 Transaction Forwarding

However, people broadcasting new location asks will often receive excess pending transactions. Fortunately, they later will have all incentives to forward each still pending one to another such broadcaster, as not doing so:

- Could reduce their future sold rights, by lessening the demand for transaction conclusions.
- Could reduce their future block-chaining rewards, by decreasing transaction volume in bytes.
- Could reduce their money's future value, by extending the wait for transaction conclusions.

Then, the only motivation for a location asks broadcaster to neither conclude nor forward a pending received transaction t would be to disrupt the system. However:

- There can be additional such broadcasters holding t, who most likely would still choose to either conclude or forward it.
- People can always:
  - Resend t to the broadcasters of new location asks, and even probe them for t before doing so.
  - Broadcast t.

11 Optional Centralization

Always to optimize monetary consensus, people unable or unwilling to engage in block chaining must be able to sell their loss to those then willing and independently able to do so. For example, a route r with a multi-signature address N must be able to lease its represented loss l to a rather single-signature address route, thus making it easier or even possible to sign a proof p of l. Indeed, the block eventually containing p can also include a block-chaining authorization informing:

1. The price of leasing l as a fraction of r's total earned fees.
2. The unique identifier of a second route d to receive an implicit transfer of the remainder.
3. This authorization's hash signed by the required number of N's private keys.

Then, only d's owners can use r to chain this block, by signing p with the private key to d's address. While conversely, from its total earned fees in each subsequent block, r must collect only the fraction constituting its lease price, and d only the remainder.

This algorithm provides the following benefits:

- It tends to minimize the proportion of spendable outputs not engaged in block chaining, or idle. For example, it allows charging for a multi-signature wallet W by requiring W's routes to be leased at most for the remainder of W's price.
- It lets all money owners control the degree and profitability of block-chaining centralization, by deciding:
  - ◇ Whether or not to engage in block chaining.
  - ◇ Whether or not to lease their eventually idle routes, to whom, and for which price.
- It lets people prevent any future leasing of their spendable outputs merely by remitting them to themselves. For example, it allows W's users to transfer each route for lease in W to another (single- or multi-signature) wallet they also control.

12 Forcibly Serial Chaining

Still, to increase their block-chaining odds or even disrupt the system, people could publicize different blocks chained by the same route at the same height,[14] thus arbitrarily delaying the consensual selection of a single chain. To discourage this parallel chaining:

1. The header H of each block b:
   - 1.1. Must contain a section including:
     - 1.1.1. The primary proof-of-loss data (see item 1. on page 15) for a route r.
     - 1.1.2. b's transaction ask (see page 13) and transaction-ask exclusion list (again on page 13).
     - 1.1.3. The hash of b's parent.
     - 1.1.4. The hash-tree root T of all (explicit) transactions in b.
     - 1.1.5. r's optional block-chaining authorization (see page 18), along with all of b's remaining data not affecting T.
   - 1.2. Can include another block's header D.
   - 1.3. Must contain b's authentication, consisting of H's — which is then also b's — hash signed by the private key to either r's or r's authorized route's address.
2. The mandatory section of D:
   - 2.1. Must have the same block height as b's parent p.
   - 2.2. Must reward the same route as p does.
3. D must differ from p's header.
4. If H includes D, then r must inherit p's total chaining reward and collect all seized fees in p (see page 11).

This way, to avoid having all their latest earnings in both rights and fees inherited by others, people will always tend to engage in serial rather than parallel block chaining.

[14] A similar vulnerability affects proof-of-stake, which — unlike proof-of-work — also needs "nothing at stake" of people's actual "wealth" (hashing-power).

13 Block Interval

Each proof of loss must inform a moment:

- Not later than the present one, or people could chain blocks in the future.
- Later than the creation time of its eventually containing block's parent, or people could chain blocks in the past.

However, people can always delay broadcasting a block, for example, to increase their block-chaining reward. Then, to prevent this delay from slowing the system by reducing total proof-of-loss possible moments hence total block-chaining odds:

1. The creation time of all blocks must be that of — the only moment informed by — their contained proof of loss, which it could not antecede.
2. The block interval must be the delay between the creation rather than broadcast times of any two consecutive blocks.

This algorithm provides the following benefits:

- It tends to prevent block-interval manipulation, by making each block's creation time depend entirely on block-chaining odds.[15]
- It tends to minimize chain-fork frequency and competing-branch length while maximizing block-interval resilience, by making total proof-of-loss possible moments hence total block-chaining odds proportional simultaneously:
    - ◇ To any delays in block broadcast or propagation.
    - ◇ To the age of the oldest current block.

[15] Then, as the (purely block-interval adjusted) general difficulty target will also depend entirely on the same block-chaining odds, the proof-of-loss context (see item 1.4. on page 15) becomes even harder to manipulate.

14 General Difficulty Target

However, to still minimize the block-chaining odds of each recently rewarded or concluded route (as already specified on page 5), the longest protocol reduction to those odds must become a proportional discount applied to the current block's deviation from its target creation time. Otherwise, if expanding, this block-interval variation would tend to restore some of those protocol-reduced odds by raising the general difficulty target, which at the current block height h ($\geqslant 0$) must hence be:

$$x_h = \begin{cases} x_{h-1} \times \left(1 + \dfrac{\dfrac{i_b}{I_b} - 1}{d_w}\right) & \text{if } h > 1 \\ x_g & \text{otherwise} \end{cases}$$

Where:

- x_h means the general difficulty target at h.
- x_{h-1} means the general difficulty target at $h - 1$.
- i_b means the actual interval between blocks at $h - 1$ and $h - 2$.
- I_b means the target interval between any two consecutive blocks.
- d_w means the block depth of the oldest reward not entirely surrendered by the route that earned it.
- x_g means the first general difficulty target.

This algorithm provides the following benefits:

- It tends to discount all deviations from the block target interval (that affect the general difficulty target) proportionally to the systemic number of routes, hence to the resilience of systemic block-chaining odds.
- It tends to exclude all privately chained branches from the longest chain proportionally to the rate of lost rights publicly engaged in block chaining, hence to system usage.

15 Consensual Chain Selection

Since proof-of-work or -stake must prove a loss they can just indicate, in both of them:

1. Indication becomes indistinguishable from proving.
2. As much as an event, each proof of an indicative sign must already be that sign, as which it must also prove what the same sign — then its proof — indicates.

Hence the need for Bitcoin's consensually selected chain to represent not only the most proof-of-work events but also the most work, and for Peercoin's one to represent not only the most proof-of-stake events but also the most stake-age destroyed, as if this way both could prove the resulting economic losses. While conversely, since unlike any such indirectly representational consensus algorithms, proof-of-loss needs no additional proof of loss, its consensually selected chain only needs to contain the most proof-of-loss events (the most blocks) during the same time by always targeting the same block interval,[16] being thus merely the longest chain.

[16] So, if any of its branches differ in their block-interval target at the same height, then their number of blocks is no longer comparable unless multiplied in each of them by the average such target for that branch.

16 Inactivity Fees

By chaining blocks, people are bearing the costs of maintaining the system. Hence the need for transaction fees. Indeed, since people transferring money are actively benefiting from that system, it is only fair that they refund its maintenance costs. However, money owners not making transactions are also benefiting from the same system, despite rather passively. Thus, it is equally fair that they also refund its maintenance costs, but how?

If a spendable output remains unspent after the loss following it has reached all that currently represented, then its faster-spent companions will alone have partly refunded the maintenance costs of all routes not spent since its block height, so:

- It no longer bears its maintenance costs.
- It becomes inactive.

Then, this route must no longer be spendable unless by paying additional fees. Otherwise, it could avoid refunding any fees paid to keep it spendable during its inactivity. Still, how much more money must people pay to spend an inactive route r?

Each time the loss either following r or the last such time reaches all that currently represented, r's missing transaction's bid must add to its total owed fees. Hence, to minimize price manipulation, this repeated increase in r's owed *inactivity fees* must be:

$$u_r = \tilde{f}_t \times l_r$$

Where:

- u_r means each undivided increase in r's owed maintenance costs.
- $\tilde{f}_t$ means the median bid transaction fee per byte in r's last indivisible inactivity period.[17]
- l_r means r's represented loss (as already specified on page 4).

[17] A median (see note 9 on page 11) is the most resistant statistic, although its resistance remains proportional to its data sample, hence to the number of transaction-fee bids concluded in that period.

Finally, for consisting only of sufficiently deferred transaction fees, all inactivity fees must be both paid and collected as if still buying the right to transact. Hence, they must always be:

- Paid by a fraction of their incurring transaction's bid, so this bid cannot be less than the combined inactivity fees owed by all inputs to its containing transaction.
- Collected by all the current block's asks as paying any otherwise unpaid part of their charged fees.

This algorithm provides the following benefits:

- Even if any balances or transaction histories are secret,[18] it causes all irretrievable balances (all unspent "dust") eventually to become publicly spent, then subject to pruning if holding no rights. Indeed, regardless of balances or transaction histories, a route r concluded at a block height h will become publicly spent once all fees currently owed by all routes at h (including r) plus all since paid by those excluding r correspond at least to h's total money supply plus all implicit earnings by r.
- Spending a long-unspent output becomes less likely to cause a subsequent spike in money velocity, as the total inactivity fees paid by this route:
    - ◇ Will have increased proportionally to its total inactivity.
    - ◇ Will disperse among all the current block's asks by being part of each one's collected bids.
- Trying to increase block-chaining odds by accruing spendable outputs will cost proportionally to the resulting gains, as the combined inactivity fees owed by all thus (im-)mobilized routes will keep often increasing.
- Each pending transaction will eventually expire, once all its owed inactivity fees have exceeded its bid.

[18] In proof-of-loss as in proof-of-stake, block-chaining decentralization requires a minimum rate and integrity of spendable-output private — hence optionally secret — ownership. However, unlike in proof-of-stake, in proof-of-loss, this route-possession privacy can always benefit from optionally secret balances (which in proof-of-stake exclude stakes) and transaction histories.

- To at least regain their paid inactivity fees if not earn additional ones, people will be more likely to engage in block chaining.
- To increase their competitiveness as transaction-right sellers, people obtaining unusually large gains from inactivity fees will tend to reduce — although by less than those excess earnings — their asked price for the right to transact.
- To reduce their paid inactivity fees, people will tend to:
  1. Unify their unspent outputs, thus minimizing the required size in bytes of the longest chain, by trying to:
     - Consolidate their incoming transactions — for example, with payment channels.
     - Avoid fragmenting their balance for other purposes than spending it.
     - Combine their already fragmented earnings that tend to remain unspent, thus reducing the number of transaction outputs not subject to pruning.
  2. Lend their otherwise unused money, thus letting other people spend it productively.

17 Intrinsic Checkpoints

Chaining a block is voting on its parent as belonging to the longest chain. Thus, if the combined loss L represented by all spendable outputs at a block height h no longer exceeds that chaining all descendants of a block b at h, then L already had the opportunity to vote or not on b, so b's descendants:

1. Must represent L's vote on b as belonging to the longest chain.
2. Must make b unalterable, or a checkpoint, since L's vote is that of all spendable outputs at h.

This algorithm provides the following benefits:

- It tends to keep the chain section above the current checkpoint as lengthy as needed, for making its length proportional both directly to the total represented loss immediately below it and inversely to the average loss represented by all its rewarded routes, hence directly to block-chaining decentralization.
- For a privately chained branch B to then replace its competing section of the consensually selected chain C, people chaining B must either:
  1. Keep it above the current checkpoint until published, when it will hence most likely still be shorter than that section.
  2. Target only people at least partly unaware of C, by providing them with a B that:
     - Includes as many transactions from C as possible, for these people to most likely accept B.
     - Lacks each transaction t from C that B's additional ones must expire by deferring (see page 24), along with any spends directly or indirectly depending on t.

  So B will tend to be rejected.
- It allows pruning the chain section simultaneously below the current checkpoint and the oldest routes with any rights, money, or affecting monetary policy or the proof-of-loss context.
- It allows each protocol change (like a younger first block or a different block target interval) to depend on most blocks above the current checkpoint voting in its favor.

18 Adaptive Monetary Policy

Not even a decentralized monetary system can operate without a monetary policy, which hence must be part of its design. Indeed, there is no decentralized way of defining general rules for creating or destroying money other than recognizing their necessity. Then, a decentralized monetary policy must result from its justification, which in proof-of-loss is as follows:

1. For prices to remain stable, the money supply must always change proportionally to its demand, thus independently of all transaction outputs no longer spendable, which cannot represent people's current demand for money.
2. Only the number of routes could represent how much money people currently need since:
   - Monetary balances cannot create that demand, for already being the actual money supply.
   - Lost rights cannot do it either, for only demanding enough money to pay their price in transaction or inactivity fees.
   - The number of spendable outputs at each block height must be at least that of their independent owners, who are collectively the source of all demand for money.
   - Eventually paid inactivity fees will always provide enough incentive to minimize the number of spendable outputs (as already specified on page 25).
3. Only still active routes could represent the actual monetary demand, of which their inactivity can only be the possibility.

Finally, this policy must always:

- Minimize the age of all transaction outputs then determining the money supply, to keep money creation or destruction as timely as possible.
- Avoid any extra rewards for concluding those active routes, to prevent the otherwise earned money from becoming an incentive to create it.

So, if the total number of monetary units is not that of all active spendable outputs at the smallest block depth with all its rights already surrendered, then the money supply must either expand or contract to make these two numbers coincide. Still:

- The limits to its expansion and contraction must be the same, to avoid unduly prioritizing either variation.
- For being a monetary cost, its contraction must not exceed all paid inactivity fees, which are then the only price paid for an oversupply of money.

Hence, the money supply:

1. Can contract at most by all paid inactivity fees, which must alone cause this contraction by fractionally disappearing.
2. Can expand at most by all paid inactivity fees, which must alone cause this expansion by fractionally doubling.

Then, at the current block height c, the systemic monetary surplus or deficit implicitly transferred to each route r must be:

$$V_r = V_c \times \frac{F_r}{F_c}$$

Where:

- V_r means all money-supply variation transferred to r at c.
- V_c means all money-supply variation at c.
- F_r means all transaction-fee bids designated to r at c.
- F_c means all transaction-fee bids at c.

So money creation or destruction can only happen the first time all rewarded routes at each block height h have no rights earned at or below h left. Then, the money supply expands or contracts until its total number of units becomes that of all active routes at h, although never by more than all currently paid inactivity fees.

This algorithm provides the following benefits:

- For being then randomly distributed, the proceeds of money creation will most likely not be an incentive to increase the number of spendable outputs per transaction.
- The money supply will tend to be always free from its arbitrary manipulation, which will hardly reward its authors with more money than it has cost them.
- Changes in monetary demand or even those in the velocity of money circulation are then unlikely to cause price volatility, which will tend to originate mostly from changes in:
  - The demand for priced items proportionally to their supply.
  - The productivity of any processes creating that supply.
- For then resulting only from an increased route activity, all newly created money will be just profits, instead of also the revenue for paying mining expenses as in Bitcoin. So none of it will tend to cause price volatility by being readily spent.
- Transaction fees need no longer transition from complementary to essential as in Bitcoin, with all associated risks. Instead, they will always be the primary reward, by at least reaching yet most likely exceeding — for including inactivity fees — and being more predictable than all simultaneously created money.
- Monetary policy becomes necessary and adaptive instead of remaining arbitrary and unresponsive as in Bitcoin, by:
  - Targeting the number of all active routes at the smallest block depth with no rights left.
  - Transferring to each currently spendable output a systemic monetary surplus or deficit proportional and limited to all inactivity fees then proportionally destined to this route.

So the money supply can grow from a single unit to any size and back with both its magnitude and allocation being minimally affected by people's wealth.

Block Validation Pseudocode

```
SYNCHRONIZED()
if not DECLARED($h_c$) or not DECLARED($validated$)
  then return false
if $h_c$ = NIL or $validated$ = NIL
  then return false
return $validated$ and
$h_c \geqslant$ GET_CURRENT_HEIGHT_FROM_PEERS()
```

```
VALIDATE_CHAIN()
INITIALIZE()
label ERROR
UNDO_UNCOMMITED_PERSISTS_AND_REMOVES()
while ACCEPTING_CANDIDATE_BLOCKS()
do $b \leftarrow$ READ_NEWLY_RECEIVED_BLOCK()
   if VALID_CANDIDATE_BLOCK($b$)
     then if READ_BLOCK_HEIGHT($b$) $> h_c$
            then VALIDATE_BLOCK($b$)
            else VALIDATE_BRANCH($b$)
          PRUNE_ORPHANED_BRANCHES()
```

```
INITIALIZE()
global $branch\_start \leftarrow chainer \leftarrow collectors \leftarrow$ NIL
global $concluding\_losses \leftarrow current\_time \leftarrow D_q \leftarrow F_b \leftarrow$ NIL
global $F_i \leftarrow h_c \leftarrow highest\_start \leftarrow h_k \leftarrow h_s \leftarrow h_w \leftarrow I_b \leftarrow$ NIL
global $implicit\_balance \leftarrow input\_depths \leftarrow inputs \leftarrow$ NIL
global $lessee \leftarrow loss\_decrease \leftarrow loss\_height \leftarrow M_c \leftarrow$ NIL
global $M_v \leftarrow open\_asks \leftarrow quorum \leftarrow quorum\_start \leftarrow$ NIL
global $transaction\_hashes \leftarrow validated \leftarrow voters \leftarrow X_c \leftarrow$ NIL
global $X_s \leftarrow x_g \leftarrow Z_k \leftarrow Z_t \leftarrow$ NIL
```

```
VALID_CANDIDATE_BLOCK(block)
 1  if not VALID_BLOCK_FORMAT(block) or
 2     not VALID_BLOCK_HASH(block)
 3     then return false
 4  SET_GLOBAL_STATE()
 5  if READ_BLOCK_TIME(block) $>$ current_time
 6     then return false
 7  if not NEW_OR_OPTIONAL_BLOCK(block)
 8     then return false
 9  if READ_BLOCK_HEIGHT(block) $= h_S$
10     then return SET_DUMMY_PARENT(block)
11  return HAS_VALID_PARENT(block)
```

```
VALID_BLOCK_HASH(block)
 1  $N \leftarrow$ GET_HASH(GET_BLOCK_HEADER_DATA(block))
 2  return READ_BLOCK_HASH(block) $= N$
```

```
SET_GLOBAL_STATE(block)
 1  if NIL $= (h_S \leftarrow$ RETRIEVE_STARTING_HEIGHT()) or
 2     NIL $= (h_C \leftarrow$ RETRIEVE_CURRENT_HEIGHT()) or
 3     NIL $=$ (validated $\leftarrow$ RETRIEVE_VALIDATED())
 4     then $h_S \leftarrow 1$
 5          $h_C \leftarrow 0$
 6          validated $\leftarrow$ false
 7  $I_b \leftarrow$ BLOCK_TARGET_INTERVAL
 8  current_time $\leftarrow$ GET_CURRENT_TIME()
 9  branch_start $\leftarrow$ READ_BLOCK_HEIGHT(block)
10  quorum $\leftarrow$ false
```

```
NEW_OR_OPTIONAL_BLOCK(block)
 1  if $h_S > (h \leftarrow$ READ_BLOCK_HEIGHT(block)$)$
 2     then return false
 3  if $h = h_S$
 4     then return $h_S > h_C$
 5  if $(h - h_C) > 1$
 6     then if $h_S > h_C$
 7             then $h_S \leftarrow h$
 8                  $h_C \leftarrow (h - 1)$
 9                  return true
10          return false
11  if RETRIEVE_BLOCK(GET_BLOCK_IDENTIFIER(block)) $\neq$ NIL
12     then return false
13  if IMMUTABLE_HEIGHT($h$)
14     then return IMMUTABLE_BLOCK(block)
15  siblings $\leftarrow$ RETRIEVE_BLOCKS_AT_HEIGHT($h$)
16  $r \leftarrow$ READ_CHAINER_IDENTIFIER(block)
17  another $\leftarrow$ false
18  for each sibling in siblings
19  do if READ_CHAINER_IDENTIFIER(sibling) $= r$
20        then if $h < h_C$ or another
21                then return false
22             another $\leftarrow$ true
23  return true
```

```
GET_BLOCK_IDENTIFIER(block)
1  $N \leftarrow$ READ_BLOCK_HASH(block)
2  height $\leftarrow$ READ_BLOCK_HEIGHT(block)
3  return CREATE_BLOCK_IDENTIFIER($N$, height)
```

```
IMMUTABLE_HEIGHT(height)
1  $D \leftarrow$ RETRIEVE_DUMMY_BLOCK()
2  $blocks \leftarrow$ READ_IMMUTABLE_BLOCKS($D$)
3  for each $X_b$ in $blocks$
4  do if $height =$ GET_BLOCK_HEIGHT($X_b$)
5          then return true
6  return false
```

```
IMMUTABLE_BLOCK(block)
1  $D \leftarrow$ RETRIEVE_DUMMY_BLOCK()
2  $X_b \leftarrow$ GET_BLOCK_IDENTIFIER($block$)
3  return EXISTS_IN_LIST(READ_IMMUTABLE_BLOCKS($D$), $X_b$)
```

```
SET_DUMMY_PARENT(block)
1  $X_b \leftarrow$ GET_BLOCK_IDENTIFIER($block$)
2  if NIL $\neq$ ($D \leftarrow$ GET_RESTORABLE_DUMMY_FROM_PEERS($X_b$))
3     then if VALID_DUMMY_PARENT($D$, $block$)
4              then PERSIST_DUMMY_BLOCK($D$)
5                   PERSIST_STARTING_HEIGHT($h_S$)
6                   PERSIST_VALIDATED(false)
7                   return true
8  return false
```

```
Valid_Dummy_Parent($D$, $block$)
if not Valid_Dummy_Format($D$) or
  not Valid_Dummy_Hash($D$)
  then return false
if Read_Dummy_Time($D$) $\geqslant$ Read_Block_Time($block$)
  then return false
if Get_Length($blocks \leftarrow$ Read_Immutable_Blocks($D$)) $= 0$
  then return false
$h \leftarrow$ Read_Block_Height($block$)
$heights \leftarrow \{\}$
for each $X_b$ in $blocks$
do if $h > (height \leftarrow$ Get_Block_Height($X_b$))
      then return false
   if $height = h$ and Get_Block_Identifier($block$) $\neq X_b$
      then return false
   Add_to_List($heights$, $height$)
$heights \leftarrow$ Order_List($heights$)
for each $height$ in $heights$
do if $height \neq h$
      then return false
   $h \leftarrow (h + 1)$
return true
```

```
Valid_Dummy_Hash($D$)
$N \leftarrow$ Get_Hash(Get_Dummy_Data($D$))
return Read_Dummy_Hash($D$) $= N$
```

```
Has_Valid_Parent($block$)
if nil $= (parent \leftarrow$ Retrieve_Parent($block$))
  then return false
$time \leftarrow$ Read_Block_Time($block$)
return Read_Block_Time($parent$) $< time$
```

```
RETRIEVE_PARENT(block)
 1  $X_b \leftarrow$ GET_PARENT_IDENTIFIER(block)
 2  return RETRIEVE_BLOCK($X_b$)
```

```
GET_PARENT_IDENTIFIER(block)
 1  $N \leftarrow$ READ_PARENT_HASH(block)
 2  $height \leftarrow$ READ_BLOCK_HEIGHT(block)
 3  return CREATE_BLOCK_IDENTIFIER($N$, $height - 1$)
```

```
VALIDATE_BLOCK(block)
  1  if not IMMUTABLE_BLOCK(block)
  2     then SET_CHAIN_STATE(block)
  3          VALIDATE_BRANCH_LENGTH(block)
  4          VALIDATE_CHAINER_ASK(block)
  5          VALIDATE_OPTIONAL_HEADER(block)
  6          VALIDATE_PROOF-OF-LOSS(block)
  7          VALIDATE_LEASE(block)
  8          VALIDATE_BLOCK_SIGNATURE(block)
  9          VALIDATE_BLOCK_TRANSACTIONS(block)
 10          VALIDATE_BLOCK_EARNINGS(block)
 11          VALIDATE_GIFT_RIGHTS(block)
 12          VALIDATE_CHAIN_PRUNING(block)
 13          VALIDATE_CHAIN_STATE(block)
 14  UPDATE_CHAIN(block)
```

```
SET_CHAIN_STATE(block)
 1  $parent \leftarrow$ RETRIEVE_PARENT(block)
 2  $x_g \leftarrow$ READ_GENERAL_MAX_HASH(parent)
 3  if $h_s > (h_w \leftarrow$ READ_ASKS_LOW(parent)$)$
 4     then ERROR(CONFLICTING_ASKS_LOW)
 5  if $(h_s - 1) > (h_k \leftarrow$ READ_CHECKPOINT(parent)$)$
 6     then ERROR(CONFLICTING_CHECKPOINT)
 7  $M_c \leftarrow$ READ_MONEY_SUPPLY(parent)
 8  $M_v \leftarrow 0$
```

```
ERROR(code)
 1  DISPLAY(GET_ERROR_MESSAGE(code))
 2  goto ERROR
```

```
VALIDATE_BRANCH_LENGTH(block)
 1  if $branch\_start \leqslant h_k$
 2     then ERROR(BRANCH_NOT_ABOVE_CHECKPOINT)
```

```
VALIDATE_CHAINER_ASK(block)
 1  if READ_ASKED_FEE_per_BYTE(block) $< 0$
 2     then ERROR(NEGATIVE_ASK)
```

```
VALIDATE_OPTIONAL_HEADER(block)
  1  if NIL $=$ ($H \leftarrow$ READ_OPTIONAL_HEADER(block))
  2     then return
  3  if NIL $\neq$ ($lease \leftarrow$ READ_HEADER_LEASE(H))
  4     then VALIDATE_OPT-HEADER_LEASE(lease, H, block)
  5           return
  6  $parent \leftarrow$ RETRIEVE_PARENT(block)
  7  $r \leftarrow$ READ_CHAINER_IDENTIFIER(parent)
  8  VALIDATE_OPT-HEADER_CHAINER(H, r)
  9  $output \leftarrow$ GET_CONCLUDED_OUTPUT(r, parent)
 10  VALIDATE_OPT-HEADER_IDENTIFIER(H, block, output)
```

```
VALIDATE_OPT-HEADER_LEASE(lease, H, block)
 1  VALIDATE_LEASE_PRICE(lease)
 2  parent ← RETRIEVE_PARENT(block)
 3  r ← READ_CHAINER_IDENTIFIER(parent)
 4  if $r = (r_l$ ← READ_LESSEE_IDENTIFIER(lease))
 5     then ERROR(LEASE_TO_OPTIONAL_SELF)
 6  if NIL $\neq$ (output ← GET_CONCLUDED_OUTPUT($r_l$, parent))
 7     then VALIDATE_OPT-HEADER_CHAINER(H, r)
 8          VALIDATE_OPT-HEADER_IDENTIFIER(H, block, output)
 9     else if not LEASE-ONLY_OPT-HEADER(H)
10             then ERROR(UNNECESSARY_OPT-HEADER_DATA)
11  N ← GET_HASH(GET_LEASE_DATA(lease))
12  if READ_LEASE_HASH(lease) $\neq$ N
13     then ERROR(INVALID_OPT-HEADER_LEASE_HASH)
14  S ← GET_LEASE-SIGNING_DATA(lease)
15  output ← GET_CONCLUDED_OUTPUT(r, parent)
16  if not VALID_SIGNATURE(output, N, S)
17     then ERROR(INVALID_OPT-HEADER_LEASE_SIGNATURE)
```

```
VALIDATE_LEASE_PRICE(lease)
 1  price ← READ_LEASE_PRICE(lease)
 2  if $price < 0$ or $price > 1$
 3     then ERROR(INVALID_LEASE_PRICE)
```

```
GET_CONCLUDED_OUTPUT(r, block)
 1  h ← READ_BLOCK_HEIGHT(block)
 2  while $h_S \leqslant (h \leftarrow (h - 1))$
 3  do block ← RETRIEVE_PARENT(block)
 4     if NIL $\neq$ (output ← GET_OUTPUT(r, block))
 5        then return output
 6  return NIL
```

```
GET_OUTPUT(r, block)
1 t ← GET_TRANSACTION_IDENTIFIER(r)
2 if NIL ≠ (transaction ← GET_TRANSACTION(t, block))
3    then n ← GET_OUTPUT_INDEX(r)
4         i ← 0
5         for each output in transaction
6         do if n = (i ← (i + 1))
7               then return output
8 return NIL
```

```
GET_TRANSACTION(t, block)
1 for each transaction in block
2 do if GET_IDENTIFIER(transaction) = t
3       then return transaction
4 return NIL
```

```
GET_IDENTIFIER(transaction)
1 N ← READ_HASH(transaction)
2 time ← READ_TIME(transaction)
3 return CREATE_TRANSACTION_IDENTIFIER(N, time)
```

```
VALIDATE_OPT-HEADER_CHAINER(H, r)
1 if READ_BLOCK-CHAINER_IDENTIFIER(H) ≠ r
2    then ERROR(INVALID_OPT-HEADER_CHAINER)
```

```
VALIDATE_OPT-HEADER_IDENTIFIER($H$, $block$, $output$)
 1  $height \leftarrow$ READ_HEADER_HEIGHT($H$)
 2  if $height \neq$ (READ_BLOCK_HEIGHT($block$) $- 1$)
 3     then ERROR(INVALID_OPT-HEADER_HEIGHT)
 4  $N \leftarrow$ GET_HASH(GET_HEADER_DATA($H$))
 5  if READ_HEADER_HASH($H$) $\neq N$
 6     then ERROR(INVALID_OPT-HEADER_HASH)
 7  if $N =$ READ_PARENT_HASH($block$)
 8     then ERROR(INVALID_OPTIONAL_HEADER)
 9  $S \leftarrow$ GET_HEADER-SIGNING_DATA($H$)
10  if not VALID_SIGNATURE($output$, $N$, $S$)
11     then ERROR(INVALID_OPT-HEADER_SIGNATURE)
```

```
VALIDATE_PROOF-OF-LOSS(block)
$h \leftarrow height \leftarrow$ READ_BLOCK_HEIGHT($block$)
$r \leftarrow$ READ_CHAINER_IDENTIFIER($b \leftarrow block$)
$descending \leftarrow \{\}$
$chainer \leftarrow b_w \leftarrow X_b \leftarrow$ NIL
$b_i \leftarrow 0$
$implicit\_balance \leftarrow$ false
while $chainer =$ NIL and $h_s \leqslant (h \leftarrow (h - 1))$
do if $X_b \neq$ NIL
        then ADD_TO_LIST($descending$, $X_b$)
    $X_b \leftarrow$ GET_PARENT_IDENTIFIER($child \leftarrow b$)
    $b \leftarrow$ RETRIEVE_BLOCK($X_b$)
    if GET_INPUT($r$, $b$) $\neq$ NIL
        then ERROR(TRANSFERRED_CHAINER)
    if READ_CHAINER_IDENTIFIER($b$) $= r$ and $b_w =$ NIL
        then $b_w \leftarrow b$
    if not $implicit\_balance$
        then $b_i \leftarrow$ GET_IMPLICIT_BALANCE($r$, $b$, $child$)
    $chainer \leftarrow$ GET_OUTPUT($r$, $b$)
if $chainer =$ NIL
    then ERROR(NONEXISTENT_CHAINER)
if $b_w =$ NIL
    then $b_w \leftarrow b$
$b_e \leftarrow$ GET_EXPLICIT_BALANCE($chainer$)
$l_r \leftarrow$ GET_ROUTE_LOSS($r$, $b$)
$ascending \leftarrow$ REVERT_LIST($descending$)
VALIDATE_ROUTE($b_e$, $b_i$, $l_r$, $ascending$)
$W_r \leftarrow$ GET_REWARD($b_w$, NIL, $l_r$, false)
$d_w \leftarrow (height -$ READ_BLOCK_HEIGHT($b_w$))
$d_r \leftarrow (height -$ READ_BLOCK_HEIGHT($b$))
$x_r \leftarrow$ GET_MAX_HASH($W_r$, $l_r$, $d_r$, $d_w$)
VALIDATE_PROOF-OF-LOSS_HASH($block$, $x_r$)
```

```
GET_INPUT(r, block)
 1  for each transaction in block
 2  do for each input in transaction
 3      do if GET_ROUTE_IDENTIFIER(input) = r
 4             then return input
 5  return NIL
```

```
GET_ROUTE_IDENTIFIER(input)
 1  $N \leftarrow$ READ_TRANSACTION_HASH(input)
 2  $time \leftarrow$ READ_TRANSACTION_TIME(input)
 3  $t \leftarrow$ CREATE_TRANSACTION_IDENTIFIER(N, time)
 4  $n \leftarrow$ READ_OUTPUT_INDEX(input)
 5  return CREATE_ROUTE_IDENTIFIER(t, n)
```

```
GET_IMPLICIT_BALANCE(r, block, child)
  1  $earnings \leftarrow$ READ_EARNINGS_PER_OUTPUT(block)
  2  $r_w \leftarrow$ GET_PROXY_CHAINER(block)
  3  $r_l \leftarrow$ GET_PROXY_LESSEE(block)
  4  $seized \leftarrow$ HAS_TWO_HEADERS(child)
  5  for each output_quota in earnings
  6  do if $r = (r_q \leftarrow$ READ_EARNER_IDENTIFIER(output_quota))
  7         then implicit_balance $\leftarrow$ true
  8              if seized and $(r_q = r_w$ or $r_q = r_l)$
  9                 then return 0
 10              return READ_OUTPUT_EARNINGS(output_quota)
 11  return 0
```

```
GET_PROXY_CHAINER(r, block)
 1  $r \leftarrow$ READ_CHAINER_IDENTIFIER(block)
 2  return GET_LAST_PROXY(r, block)
```

```
GET_LAST_PROXY($r$, $block$)
1  while NIL $\neq$ ($input \leftarrow$ GET_INPUT($r$, $block$))
2  do if NIL $=$ ($r \leftarrow$ GET_PROXY_IDENTIFIER($r$, $block$))
3        then ERROR(MISSING_PROXY_ROUTE)
4  return $r$
```

```
GET_PROXY_IDENTIFIER($r$, $block$)
1  $p \leftarrow$ GET_BLOCK_PROXIES($block$)
2  return READ_PROXY_IDENTIFIER($p$, $r$)
```

```
GET_BLOCK_PROXIES($block$)
1  $X_b \leftarrow$ GET_BLOCK_IDENTIFIER($block$)
2  if NIL $=$ ($p \leftarrow$ RETRIEVE_PROXY_MAP($X_b$))
3    then $p \leftarrow$ CREATE_PROXY_MAP($block$)
4         PERSIST_PROXY_MAP($X_b$, $p$)
5  return $p$
```

```
CREATE_PROXY_MAP($block$)
1  $p \leftarrow$ CREATE_EMPTY_PROXY_MAP()
2  for each $transaction$ in $block$
3  do for each $input$ in $transaction$
4       do if NIL $\neq$ ($n \leftarrow$ READ_PROXY-ROUTE_INDEX($input$))
5            then SET_PROXY($p$, $n$, $input$, $transaction$)
6  return $p$
```

```
SET_PROXY($p$, $n$, $input$, $transaction$)
  $r \leftarrow$ GET_ROUTE_IDENTIFIER($input$)
  $t \leftarrow$ GET_IDENTIFIER($transaction$)
  $i \leftarrow 0$
  for each $output$ in $transaction$
  do if $n = (i \leftarrow (i + 1))$
        then if READ_EXPLICIT_BALANCE($output$) $= 0$
                then ERROR(PROVABLY_UNSPENDABLE_PROXY)
             $r_p \leftarrow$ CREATE_ROUTE_IDENTIFIER($t$, $n$)
             WRITE_PROXY_IDENTIFIER($p$, $r$, $r_p$)
             return
  ERROR(NONEXISTENT_PROXY_ROUTE)
```

```
GET_PROXY_LESSEE($r$, $block$)
  if NIL $\neq$ ($lease \leftarrow$ READ_LEASE($block$))
     then $r \leftarrow$ READ_LESSEE_IDENTIFIER($lease$)
          return GET_LAST_PROXY($r$, $block$)
  return NIL
```

```
GET_EXPLICIT_BALANCE($output$)
  if $0 = (b_e \leftarrow$ READ_EXPLICIT_BALANCE($output$))
     then ERROR(PROVABLY_UNSPENDABLE_OUTPUT)
  return $b_e$
```

```
GET_ROUTE_LOSS($r$, $block$)
  $t \leftarrow$ GET_TRANSACTION_IDENTIFIER($r$)
  $transaction \leftarrow$ GET_TRANSACTION($t$, $block$)
  $n \leftarrow$ GET_OUTPUT_INDEX($r$)
  return GET_LOSS($n$, $transaction$)
```

```
GET_LOSS($n$, $transaction$)
 1  $z_t \leftarrow$ GET_SIZE_IN_BYTES($transaction$)
 2  $z_r \leftarrow Z_r \leftarrow i \leftarrow 0$
 3  for each $output$ in $transaction$
 4  do $i \leftarrow (i + 1)$
 5      if READ_EXPLICIT_BALANCE($output$) $> 0$
 6         then $Z_r \leftarrow (Z_r + (z \leftarrow$ GET_SIZE_IN_BYTES($output$)))
 7                 if $n = i$
 8                    then $z_r \leftarrow z$
 9  return $z_t \times (z_r \div Z_r)$
```

```
VALIDATE_ROUTE($b_e$, $b_i$, $l_r$, $ascending$)
 1  if GET_INACTIVITY_FEES($l_r$, $ascending$) $\geqslant (b_e + b_i)$
 2     then ERROR(SPENT_OUTPUT)
```

```
GET_INACTIVITY_FEES($l_r$, $ascending$)
  1  $losses \leftarrow F_t \leftarrow 0$
  2  $bids \leftarrow \{\}$
  3  for each $X_b$ in $ascending$
  4  do $block \leftarrow$ RETRIEVE_BLOCK($X_b$)
  5      $L_c \leftarrow$ READ_CURRENT_LOSS($block$)
  6      for each $transaction$ in $block$
  7      do $losses \leftarrow (losses +$ GET_SIZE_IN_BYTES($transaction$))
  8          $f_t \leftarrow$ READ_BID_FEE_PER_BYTE($transaction$)
  9          ADD_TO_LIST($bids$, $f_t$)
 10      if $losses \geqslant L_c$
 11         then $F_t \leftarrow (F_t +$ GET_MEDIAN($bids$))
 12                $losses \leftarrow 0$
 13                $bids \leftarrow \{\}$
 14  return $F_t \times l_r$
```

```
GET_MEDIAN(list)
1 $list \leftarrow$ ORDER_LIST(list)
2 $remainder \leftarrow ((length \leftarrow$ GET_LENGTH(list)) mod 2)
3 $start \leftarrow$ (TRIM_TO_INTEGER($length \div 2$) + $remainder$)
4 $start \leftarrow$ (LISTS_FIRST_INDEX + ($start - 1$))
5 if $remainder = 1$
6   then return READ_ELEMENT(list, start)
7 $e_1 \leftarrow$ READ_ELEMENT(list, start)
8 $e_2 \leftarrow$ READ_ELEMENT(list, start + 1)
9 return $(e_1 + e_2) \div 2$
```

```
GET_REWARD(block, child, $l_r$, seizing)
 1 if $child \neq$ NIL
 2   then if HAS_TWO_HEADERS(child)
 3          then return 0
 4 if READ_BLOCK_HEIGHT(block) $> h_s$
 5   then parent $\leftarrow$ RETRIEVE_PARENT(block)
 6        if seizing and HAS_TWO_HEADERS(block)
 7          then $l \leftarrow$ GET_CHAINER_LOSS(parent)
 8               $W_r \leftarrow$ GET_REWARD(parent, NIL, $l$, true)
 9          else $W_r \leftarrow$ READ_GIFT_RIGHTS(parent)
10   else $D \leftarrow$ RETRIEVE_DUMMY_BLOCK()
11        if seizing and HAS_TWO_HEADERS(block)
12          then $W_r \leftarrow$ READ_DUMMY_REWARD($D$)
13          else $W_r \leftarrow$ READ_DUMMY_GIFT_RIGHTS($D$)
14 for each transaction in block
15 do $W_r \leftarrow (W_r$ + GET_SIZE_IN_BYTES(transaction))
16 return $W_r + l_r$
```

```
GET_CHAINER_LOSS(block)
 1  $r \leftarrow$ READ_CHAINER_IDENTIFIER(block)
 2  $t \leftarrow$ GET_TRANSACTION_IDENTIFIER($r$)
 3  $h \leftarrow$ READ_BLOCK_HEIGHT($b \leftarrow block$)
 4  transaction $\leftarrow$ NIL
 5  while transaction = NIL and $h_s \leqslant (h \leftarrow (h - 1))$
 6  do $b \leftarrow$ RETRIEVE_PARENT($b$)
 7     transaction $\leftarrow$ GET_TRANSACTION($t$, $b$)
 8  if transaction $\neq$ NIL
 9     then loss_height $\leftarrow$ READ_BLOCK_HEIGHT($b$)
10          $n \leftarrow$ GET_OUTPUT_INDEX($r$)
11          return GET_LOSS($n$, transaction)
12  $D \leftarrow$ RETRIEVE_DUMMY_BLOCK()
13  $X_b \leftarrow$ GET_BLOCK_IDENTIFIER(block)
14  if NIL = ($l_r \leftarrow$ READ_CHAINER_LOSS($D$, $X_b$))
15     then ERROR(MISSING_CHAINER_LOSS)
16  if NIL = (loss_height $\leftarrow$ READ_LOSS_HEIGHT($D$, $X_b$))
17     then ERROR(MISSING_CHAINER-LOSS_HEIGHT)
18  return $l_r$
```

```
GET_MAX_HASH($W_r$, $l_r$, $d_r$, $d_w$)
 1  if $W_r < (W_s \leftarrow (d_w \times l_r))$
 2     then $W_s \leftarrow W_r$
 3  $m_1 \leftarrow ((l_r \div W_r) + ((1 - (l_r \div W_r)) \times (W_s \div W_r)))$
 4  $m_2 \leftarrow (((W_s \div l_r) + 1) \div d_r)$
 5  return $x_g \times l_r \times m_1 \times m_2$
```

```
VALIDATE_PROOF-OF-LOSS_HASH(block, $x_r$)
 1  VALIDATE_PROOF-OF-LOSS_CONTEXT(block)
 2  $N \leftarrow$ GET_HASH(GET_PROOF-OF-LOSS_DATA(block))
 3  if READ_PROOF-OF-LOSS_HASH(block) $\neq N$
 4     then ERROR(INVALID_PROOF-OF-LOSS_HASH)
 5  if $N > x_r$
 6     then ERROR(LOW_PROOF-OF-LOSS_DIFFICULTY)
```

```
VALIDATE_PROOF-OF-LOSS_CONTEXT(block)
 1  $h \leftarrow height \leftarrow$ READ_BLOCK_HEIGHT($b \leftarrow block$)
 2  while $h_w \leqslant (h \leftarrow (h - 1))$
 3  do $b \leftarrow$ RETRIEVE_PARENT($b$)
 4  $l_r \leftarrow$ GET_CHAINER_LOSS($b$)
 5  $d_r \leftarrow (height - loss\_height)$
 6  $r \leftarrow$ READ_CHAINER_IDENTIFIER($b$)
 7  $r_w \leftarrow$ NIL
 8  $b \leftarrow block$
 9  while $r_w \neq r$
10  do $b \leftarrow$ RETRIEVE_PARENT($b$)
11      $r_w \leftarrow$ READ_CHAINER_IDENTIFIER($b$)
12  $W_r \leftarrow$ GET_REWARD($b$, NIL, $l_r$, false)
13  $d_w \leftarrow (height -$ READ_BLOCK_HEIGHT($b$))
14  $x_c \leftarrow$ GET_MAX_HASH($W_r$, $l_r$, $d_r$, $d_w$)
15  if READ_PROOF-OF-LOSS_CONTEXT($block$) $\neq x_c$
16     then ERROR(INVALID_PROOF-OF-LOSS_CONTEXT)
```

```
VALIDATE_LEASE(block)
if (lessee ← NIL) = (lease ← READ_LEASE(block))
  then return
VALIDATE_LEASE_PRICE(lease)
$r$ ← READ_LESSEE_IDENTIFIER(lease)
if $r$ = READ_CHAINER_IDENTIFIER(block)
  then ERROR(LEASE_TO_SELF)
$h$ ← READ_BLOCK_HEIGHT(block)
descending ← {}
$X_b$ ← NIL
$b_i$ ← 0
implicit_balance ← false
while lessee = NIL and $h_S \leqslant (h \leftarrow (h - 1))$
do if $X_b \neq$ NIL
      then ADD_TO_LIST(descending, $X_b$)
   $X_b$ ← GET_PARENT_IDENTIFIER(child ← block)
   block ← RETRIEVE_BLOCK($X_b$)
   if GET_INPUT($r$, block) ≠ NIL
      then ERROR(TRANSFERRED_LESSEE)
   if not implicit_balance
      then $b_i$ ← GET_IMPLICIT_BALANCE($r$, block, child)
   lessee ← GET_OUTPUT($r$, block)
if lessee = NIL
  then ERROR(NONEXISTENT_LESSEE)
$b_e$ ← GET_EXPLICIT_BALANCE(lessee)
$l_r$ ← GET_ROUTE_LOSS($r$, block)
ascending ← REVERT_LIST(descending)
VALIDATE_ROUTE($b_e$, $b_i$, $l_r$, ascending)
$N$ ← GET_HASH(GET_LEASE_DATA(lease))
if READ_LEASE_HASH(lease) ≠ $N$
  then ERROR(INVALID_LEASE_HASH)
$S$ ← GET_LEASE-SIGNING_DATA(lease)
if not VALID_SIGNATURE(chainer, $N$, $S$)
  then ERROR(INVALID_LEASE_SIGNATURE)
```

```
VALIDATE_BLOCK_SIGNATURE(block)
 1  if NIL = ($output \leftarrow lessee$)
 2     then $output \leftarrow chainer$
 3  $H \leftarrow$ READ_HEADER(block)
 4  $N \leftarrow$ GET_HASH(GET_HEADER_DATA($H$))
 5  if READ_HEADER_HASH($H$) $\neq N$
 6     then ERROR(INVALID_BLOCK_HASH)
 7  $S \leftarrow$ GET_HEADER-SIGNING_DATA($H$)
 8  if not VALID_SIGNATURE($output$, $N$, $S$)
 9     then ERROR(INVALID_BLOCK_SIGNATURE)
```

```
VALIDATE_BLOCK_TRANSACTIONS(block)
  1  $Z_t \leftarrow loss\_decrease \leftarrow F_b \leftarrow F_i \leftarrow 0$
  2  FORGET_ROUTE_EARNINGS()
  3  $inputs \leftarrow \{\}$
  4  $input\_depths \leftarrow \{\}$
  5  $concluding\_losses \leftarrow \{\}$
  6  $transaction\_hashes \leftarrow \{\}$
  7  $collectors \leftarrow \{\}$
  8  for each transaction in block
  9  do VALIDATE_TRANSACTION(transaction, block)
 10  $N \leftarrow$ GET_HASH-TREE_ROOT(transaction_hashes)
 11  if READ_TRANSACTIONS_HASH-TREE_ROOT(block) $\neq N$
 12     then ERROR(INVALID_TRANSACTIONS_HASH-TREE_ROOT)
 13  $parent \leftarrow$ RETRIEVE_PARENT(block)
 14  $L_c \leftarrow$ READ_CURRENT_LOSS(parent)
 15  if READ_CURRENT_LOSS(block) $\neq (L_c + (Z_t - loss\_decrease))$
 16     then ERROR(INVALID_CURRENT_LOSS)
```

```
Validate_Transaction($transaction$, $block$)
  Validate_Transaction_Format($transaction$)
  Validate_Transaction_Identifier($transaction$, $block$)
  $B_t \gets B_e \gets B_i \gets F_r \gets 0$
  for each $output$ in $transaction$
  do $B_t \gets (B_t +$ Get_Output_Balance($output$))
  Set_Proxy_Balances($transaction$, $block$)
  $height \gets$ Read_Block_Height($block$)
  for each $input$ in $transaction$
  do $b \gets block$
      $r \gets$ Get_Route_Identifier($input$)
      $descending \gets \{\}$
      $X_b \gets$ nil
      while nil $= (output \gets$ Get_Output($r$, $b$))
      do if $X_b \neq$ nil
              then Add_to_List($descending$, $X_b$)
         $X_b \gets$ Get_Parent_Identifier($b$)
         $b \gets$ Retrieve_Block($X_b$)
      $B_e \gets (B_e + (b_e \gets$ Get_Explicit_Balance($output$)))
      $B_i \gets (B_i + (b_i \gets$ Get_Proxy_Balance($r$, $b$)))
      $l_r \gets$ Get_Route_Loss($r$, $b$)
      $ascending \gets$ Revert_List($descending$)
      $F_r \gets (F_r + (f_r \gets$ Get_Inactivity_Fees($l_r$, $ascending$)))
      Validate_Input($input$, $b_e$, $b_i$, $f_r$, $output$, $transaction$)
      $loss\_decrease \gets (loss\_decrease + l_r)$
      if $0 < (d_r \gets (height -$ Read_Block_Height($b$)))
         then Add_to_List($input\_depths$, $d_r$)
  Validate_Bid($transaction$, $B_t$, $B_e$, $B_i$, $F_r$)
```

```
VALIDATE_TRANSACTION_IDENTIFIER(transaction, block)
  $N \leftarrow$ GET_HASH(GET_TRANSACTION_DATA(transaction))
  if READ_HASH(transaction) $\neq N$
    then ERROR(INVALID_TRANSACTION_HASH)
  if EXISTS_IN_LIST(transaction_hashes, $N$)
    then ERROR(DUPLICATE_TRANSACTION_HASH)
  ADD_TO_LIST(transaction_hashes, $N$)
  time $\leftarrow$ READ_TIME(transaction)
  if time $>$ READ_BLOCK_TIME(block)
    then ERROR(LATE_TRANSACTION)
  for each input in transaction
  do if time $<$ READ_TRANSACTION_TIME(input)
        then ERROR(EARLY_TRANSACTION)
  $t \leftarrow$ CREATE_TRANSACTION_IDENTIFIER($N$, time)
  $h \leftarrow$ READ_BLOCK_HEIGHT(block)
  while $h_s \leqslant (h \leftarrow (h - 1))$
  do block $\leftarrow$ RETRIEVE_PARENT(block)
     if time $>$ READ_BLOCK_TIME(block)
        then return
     if GET_TRANSACTION($t$, block) $\neq$ NIL
        then ERROR(DUPLICATE_TRANSACTION)
```

```
GET_OUTPUT_BALANCE(output)
  if $0 > (b_e \leftarrow$ READ_EXPLICIT_BALANCE(output)$)$
    then ERROR(NEGATIVE_OUTPUT_BALANCE)
  if PROVABLY_UNSPENDABLE(output)
    then if $b_e > 0$
            then ERROR(ILLEGAL_MONEY_DESTRUCTION)
    else if $b_e = 0$
            then ERROR(NULL_SPENDABLE_BALANCE)
  return $b_e$
```

```
SET_PROXY_BALANCES(transaction, block)
1  for each input in transaction
2  do $r \leftarrow$ GET_ROUTE_IDENTIFIER(input)
3      if $0 < (b_i \leftarrow$ GET_ROUTE_BALANCE$(r, block))$
4          then SET_BALANCES(input, $b_i$, block)
```

```
GET_ROUTE_BALANCE(r, block)
 1  $h_p \leftarrow ((h \leftarrow$ READ_BLOCK_HEIGHT$(block)) - 1)$
 2  $b_i \leftarrow b_s \leftarrow 0$
 3  implicit_balance $\leftarrow$ false
 4  while NIL $= (output \leftarrow$ GET_OUTPUT$(r, block))$
 5  do if $h_s > (h \leftarrow (h - 1))$
 6          then ERROR(NONEXISTENT_OUTPUT)
 7      block $\leftarrow$ RETRIEVE_PARENT(child $\leftarrow$ block)
 8      if GET_INPUT(r, block) $\neq$ NIL
 9          then ERROR(TRANSFERRED_OUTPUT)
10      if $h = h_p$
11          then $b_s \leftarrow$ GET_SEIZED_BALANCE(r, block, child)
12      if not implicit_balance
13          then $b_i \leftarrow$ GET_IMPLICIT_BALANCE(r, block, child)
14  return $b_i + b_s$
```

```
GET_SEIZED_BALANCE(r, block, child)
 1  if not HAS_TWO_HEADERS(child) or
 2     0 = (ratio ← GET_SEIZING_RATIO(r, child))
 3     then return 0
 4  earnings ← READ_EARNINGS_PER_OUTPUT(block)
 5  $b_s$ ← 0
 6  $r_w$ ← GET_PROXY_CHAINER(block)
 7  $r_l$ ← GET_PROXY_LESSEE(block)
 8  for each output_quota in earnings
 9  do $r_q$ ← READ_EARNER_IDENTIFIER(output_quota)
10       if $r_q = r_w$ or $r_q = r_l$
11          then $b_i$ ← READ_OUTPUT_EARNINGS(output_quota)
12               $b_s$ ← ($b_s$ + ($b_i$ × ratio))
13               if $b_i > 0$ and GET_INPUT($r_q$, child) = NIL
14                  then SET_ROUTE_BALANCE($r_q$, 0)
15  return $b_s$
```

```
GET_SEIZING_RATIO(r, block)
1  ratio ← 1
2  if NIL ≠ (lease ← READ_LEASE(block))
3     then ratio ← READ_LEASE_PRICE(lease)
4          if r = READ_LESSEE_IDENTIFIER(lease)
5             then return 1 − ratio
6  if r = READ_CHAINER_IDENTIFIER(block)
7     then return ratio
8  return 0
```

```
SET_ROUTE_BALANCE(r, $b_i$)
1  MEMORIZE_ROUTE_EARNINGS(r, $b_i$)
2  if not EXISTS_IN_LIST(collectors, r)
3     then ADD_TO_LIST(collectors, r)
```

```
SET_BALANCES($input$, $b_i$, $block$)
1  $r \leftarrow$ GET_ROUTE_IDENTIFIER($input$)
2  SET_ROUTE_EARNINGS($r$, $b_i$)
3  $rp \leftarrow$ GET_PROXY_IDENTIFIER($r$, $block$)
4  while $rp \neq$ NIL and $rp \neq r$
5  do SET_ROUTE_EARNINGS($rp$, $b_i$)
6      if NIL $\neq$ ($input \leftarrow$ GET_INPUT($r \leftarrow rp$, $block$))
7         then $rp \leftarrow$ GET_PROXY_IDENTIFIER($r$, $block$)
8         else  if not EXISTS_IN_LIST($collectors$, $rp$)
9                  then ADD_TO_LIST($collectors$, $rp$)
```

```
SET_ROUTE_EARNINGS($r$, $b_i$)
1  if NIL $\neq$ ($v \leftarrow$ RECALL_ROUTE_EARNINGS($r$))
2     then $b_i \leftarrow (b_i + v)$
3  MEMORIZE_ROUTE_EARNINGS($r$, $b_i$)
```

```
GET_PROXY_BALANCE($r$, $block$)
1  if GET_PROXY_IDENTIFIER($r$, $block$) = NIL and
2     NIL $\neq$ ($b_i \leftarrow$ RECALL_ROUTE_EARNINGS($r$))
3     then return $b_i$
4  return 0
```

```
VALIDATE_INPUT($input$, $b_e$, $b_i$, $fr$, $output$, $transaction$)
 1  if $fr \geqslant (b_e + b_i)$
 2     then ERROR(SPENT_INPUT)
 3  $N \leftarrow$ READ_HASH($transaction$)
 4  $S \leftarrow$ GET_TRANSACTION-SIGNING_DATA($input$, $transaction$)
 5  if not VALID_SIGNATURE($output$, $N$, $S$)
 6     then ERROR(INVALID_TRANSACTION_SIGNATURE)
 7  $r \leftarrow$ GET_ROUTE_IDENTIFIER($input$)
 8  if EXISTS_IN_LIST($inputs$, $r$)
 9     then ERROR(DUPLICATE_INPUT)
10  ADD_TO_LIST($inputs$, $r$)
```

```
VALIDATE_BID($transaction, B_t, B_e, B_i, F_r$)
    if $B_t = 0$
      then ERROR(ROUTELESS_TRANSACTION)
    $f_t \leftarrow$ READ_BID_FEE_PER_BYTE($transaction$)
    $z_t \leftarrow$ GET_SIZE_IN_BYTES($transaction$)
    if $F_r > (bid \leftarrow (f_t \times z_t))$
      then ERROR(LOW_BID)
    if $(B_t + bid) \neq (B_e + B_i)$
      then ERROR(INEXACT_SPENDING)
    $F_b \leftarrow (F_b + bid)$
    $Z_t \leftarrow (Z_t + z_t)$
    ADD_TO_LIST($concluding\_losses, z_t$)
    $F_i \leftarrow (F_i + F_r)$
```

```
VALIDATE_BLOCK_EARNINGS($block$)
    $Z_k \leftarrow 0$
    FORGET_FAILING_TO_SELL()
    $asks \leftarrow$ GET_ASKS($block$)
    if $Z_t > Z_k$
      then ERROR(UNAUTHORIZED_LOSSES)
    $F_z \leftarrow 0$
    for each $ask$ in $asks$
    do if NIL $\neq (z_k \leftarrow$ READ_SOURCE_REWARD($ask$))
           then $k \leftarrow$ GET_SOURCE_IDENTIFIER($ask$)
                $z_b \leftarrow ((z_k \div Z_k) \times Z_t)$
                MEMORIZE_FAILING_TO_SELL($k, z_k - z_b$)
       $z_b \leftarrow$ ((READ_RIGHTS_FOR_SALE($ask$) $\div Z_k) \times Z_t$)
       $f_k \leftarrow$ READ_ASKED_FEES($ask$)
       WRITE_CHARGED_FEES($ask, f_z \leftarrow (f_k \times z_b)$)
       $F_z \leftarrow (F_z + f_z)$
    if $F_b < F_z$
      then ERROR(UNPAID_FEES)
    APPLY_PRIORITIZATION($asks, block, F_z$)
    APPLY_MONETARY_POLICY($asks, block$)
    VALIDATE_EARNINGS($block, asks$)
```

```
GET_ASKS($block$)
  $h_p \leftarrow ((h \leftarrow$ READ_BLOCK_HEIGHT$(b \leftarrow block)) - 1)$
  $asks \leftarrow \{\}$
  $open\_asks \leftarrow \{\}$
  $blocks \leftarrow \{\}$
  FORGET_REVOKED_RIGHTS()
  FORGET_CONCLUDED_LOSSES()
  $X_b \leftarrow$ NIL
  while $h_w \leqslant (h \leftarrow (h - 1))$
  do if $X_b \neq$ NIL
          then ADD_TO_LIST$(blocks,\ X_b)$
      $X_b \leftarrow$ GET_PARENT_IDENTIFIER$(child \leftarrow b)$
      $b \leftarrow$ RETRIEVE_BLOCK$(X_b)$
      $l_r \leftarrow$ GET_CHAINER_LOSS$(b)$
      $W_r \leftarrow$ GET_REWARD$(b,\ child,\ l_r,$ true$)$
      if $W_r > (W_s \leftarrow ((h_p - h) \times l_r))$
          then if $l_r < (z_k \leftarrow (W_r - W_s))$
                  then $z_k \leftarrow l_r$
               $r \leftarrow$ READ_CHAINER_IDENTIFIER$(b)$
               $k \leftarrow$ CREATE_ASK_IDENTIFIER$(h,\ r)$
               SET_REVOKED_RIGHTS$(k,\ blocks)$
               MEMORIZE_CONCLUDED_LOSS$(r,\ l_r)$
               ADD_TO_LIST$(open\_asks,\ k)$
               ADD_ASKS$(b,\ z_k,\ block,\ asks)$
  return $asks$
```

```
SET_REVOKED_RIGHTS($k,\ blocks$)
  $W_k \leftarrow 0$
  for each $X_b$ in $blocks$
  do $b \leftarrow$ RETRIEVE_BLOCK$(X_b)$
      if NIL $\neq (w_k \leftarrow$ READ_REVOKED_RIGHTS$(b,\ k))$
         then $W_k \leftarrow (W_k + w_k)$
  MEMORIZE_REVOKED_RIGHTS$(k,\ W_k)$
```

```
Add_Asks($b$, $z_k$, $block$, $asks$)
  $h \leftarrow$ Read_Block_Height($b$)
  $r \leftarrow$ Read_Chainer_Identifier($b$)
  if Exists_in_List(Read_Excluded_Ask_Heights($block$), $h$)
    then $k \leftarrow$ Create_Ask_Identifier($h$, $r$)
         Memorize_Failing_to_Sell($k$, $z_k$)
         return
  $f_k \leftarrow$ Read_Asked_Fee_per_Byte($b$)
  $Z_k \leftarrow (Z_k + (z_s \leftarrow z_k))$
  if nil $\neq$ ($lease \leftarrow$ Read_Lease($b$))
    then $r_l \leftarrow$ Read_Lessee_Identifier($lease$)
         $r_p \leftarrow$ Get_Proxy($r_l$, $block$, $h$)
         Set_Balance($r_p$, $block$)
         $k \leftarrow$ Create_Ask_Identifier($h$, $r_p$)
         $z_s \leftarrow (z_k \times (price \leftarrow$ Read_Price($lease$)))
         $ask \leftarrow$ Create_Ask($k$, $f_k$, $z_k \times (1 - price)$, nil, $r$)
         Add_to_List($asks$, $ask$)
  $r_p \leftarrow$ Get_Proxy($r$, $block$, $h$)
  Set_Balance($r_p$, $block$)
  $k \leftarrow$ Create_Ask_Identifier($h$, $r_p$)
  $ask \leftarrow$ Create_Ask($k$, $f_k$, $z_s$, $z_k$, $r$)
  Add_to_List($asks$, $ask$)
```

```
Get_Proxy($r$, $block$, $height$)
  $h \leftarrow$ Read_Block_Height($b \leftarrow block$)
  $descending \leftarrow \{\}$
  while $height \leqslant (h \leftarrow (h - 1))$
  do $X_b \leftarrow$ Get_Parent_Identifier($b$)
     Add_to_List($descending$, $X_b$)
     $b \leftarrow$ Retrieve_Block($X_b$)
  $ascending \leftarrow$ Revert_List($descending$)
  for each $X_b$ in $ascending$
  do $b \leftarrow$ Retrieve_Block($X_b$)
     $r \leftarrow$ Get_Last_Proxy($r$, $b$)
  return Get_Last_Proxy($r$, $block$)
```

```
SET_BALANCE($r$, $block$)
1  if RECALL_ROUTE_EARNINGS($r$) = NIL
2     then $b_i$ ← GET_ROUTE_BALANCE($r$, $block$)
3          SET_ROUTE_BALANCE($r$, $b_i$)
```

```
CREATE_ASK($k$, $f_k$, $z_s$, $z_k$, $r$)
1  $ask$ ← CREATE_EMPTY_ASK()
2  WRITE_ASK_IDENTIFIER($ask$, $k$)
3  WRITE_ASKED_FEES($ask$, $f_k$)
4  WRITE_RIGHTS_FOR_SALE($ask$, $z_s$)
5  WRITE_SOURCE_REWARD($ask$, $z_k$)
6  WRITE_OWNER_IDENTIFIER($ask$, $r$)
7  WRITE_CHARGED_FEES($ask$, NIL)
8  WRITE_EARNINGS($ask$, NIL)
9  return $ask$
```

```
GET_SOURCE_IDENTIFIER($ask$)
1  $h$ ← GET_HEIGHT(READ_ASK_IDENTIFIER($ask$))
2  $r$ ← READ_OWNER_IDENTIFIER($ask$)
3  return CREATE_ASK_IDENTIFIER($h$, $r$)
```

```
APPLY_PRIORITIZATION($asks$, $block$, $F_z$)
seized ← 0
if $F_z > 0$
  then earnings ← {}
       for each ask in asks
       do if NIL ≠ ($z_k$ ← READ_SOURCE_REWARD(ask))
             then $z_b \leftarrow ((z_k \div Z_k) \times Z_t)$
                  $f_z$ ← (READ_ASKED_FEES(ask) × $z_b$)
                  ADD_TO_LIST(earnings, $F_b \times (f_z \div F_z)$)
       priority ← GET_PRIORITIZATION_QUOTIENT()
       seized ← (GET_MEDIAN(earnings) × priority)
       if seized > $F_b$
         then seized ← $F_b$
       remnant ← ($F_b$ − seized)
       for each ask in asks
       do $f_z$ ← READ_CHARGED_FEES(ask)
          WRITE_EARNINGS(ask, remnant × ($f_z \div F_z$))
h ← READ_BLOCK_HEIGHT(block)
r ← READ_CHAINER_IDENTIFIER(block)
$f_s$ ← seized
if NIL ≠ (lease ← READ_LEASE(block))
  then $r_l$ ← READ_LESSEE_IDENTIFIER(lease)
       $r_p$ ← GET_LAST_PROXY($r_l$, block)
       SET_BALANCE($r_p$, block)
       k ← CREATE_ASK_IDENTIFIER(h, $r_p$)
       ask ← CREATE_ASK(k, NIL, NIL, NIL, NIL)
       $f_s$ ← (seized × (price ← READ_PRICE(lease)))
       WRITE_EARNINGS(ask, seized × (1 − price))
       ADD_TO_LIST(asks, ask)
$r_p$ ← GET_LAST_PROXY(r, block)
SET_BALANCE($r_p$, block)
k ← CREATE_ASK_IDENTIFIER(h, $r_p$)
ask ← CREATE_ASK(k, NIL, NIL, NIL, NIL)
WRITE_EARNINGS(ask, $f_s$)
ADD_TO_LIST(asks, ask)
```

```
GET_PRIORITIZATION_QUOTIENT()
1  $average\_depth \leftarrow$ GET_AVERAGE($input\_depths$)
2  $median\_depth \leftarrow$ GET_MEDIAN($input\_depths$)
3  $depth\_quotient \leftarrow (average\_depth \div median\_depth)$
4  $average\_loss \leftarrow$ GET_AVERAGE($concluding\_losses$)
5  $median\_loss \leftarrow$ GET_MEDIAN($concluding\_losses$)
6  $loss\_quotient \leftarrow (median\_loss \div average\_loss)$
7  return $depth\_quotient \times loss\_quotient$
```

```
GET_AVERAGE($list$)
1  return GET_SUM($list$) $\div$ GET_LENGTH($list$)
```

```
APPLY_MONETARY_POLICY($asks$, $block$)
 1  if $M_c = (target \leftarrow$ GET_MONETARY_TARGET($block$)$)$
 2     then return
 3  if $0 < (M_v \leftarrow (target - M_c))$
 4     then if $M_v > F_i$
 5             then $M_v \leftarrow F_i$
 6     else if $0 > (F_i + M_v)$
 7             then $M_v \leftarrow (0 - F_i)$
 8  for each $ask$ in $asks$
 9  do $f_b \leftarrow$ READ_EARNINGS($ask$)
10      WRITE_EARNINGS($ask$, $f_b + (M_v \times (f_b \div F_b))$)
```

```
Get_Monetary_Target(block)
if $h_w = h_s$
    then return $M_c$
$h \leftarrow$ Read_Block_Height(block)
$losses \leftarrow L_c \leftarrow 0$
$descending \leftarrow \{\}$
$h_m \leftarrow (h_w - 1)$
while $h \geqslant h_w$ or $losses < L_c$
do if $h = h_s$
        then Error(conflicting_losses)
    $X_b \leftarrow$ Get_Parent_Identifier(block)
    $block \leftarrow$ Retrieve_Block($X_b$)
    if $h_w > (h \leftarrow (h - 1))$
        then if $h = h_m$
                 then $L_c \leftarrow$ Read_Current_Loss(block)
             Add_to_List(descending, $X_b$)
             for each transaction in block
             do $z_t \leftarrow$ Get_Size_in_Bytes(transaction)
                $losses \leftarrow (losses + z_t)$
$ascending \leftarrow$ Revert_List(descending)
return Get_Length(Get_Routes(ascending))
```

```
Get_Routes(ascending)
$routes \leftarrow \{\}$
for each $X_b$ in ascending
do $block \leftarrow$ Retrieve_Block($X_b$)
    for each transaction in block
    do Add_Routes(routes, transaction)
        Drop_Routes(routes, transaction)
return routes
```

```
ADD_ROUTES(routes, transaction)
    $t \leftarrow$ GET_IDENTIFIER(transaction)
    $n \leftarrow 0$
    for each output in transaction
    do $n \leftarrow (n + 1)$
        if READ_EXPLICIT_BALANCE(output) $> 0$
            then $r \leftarrow$ CREATE_ROUTE_IDENTIFIER($t$, $n$)
                 ADD_TO_LIST(routes, $r$)
```

```
DROP_ROUTES(routes, transaction)
    for each input in transaction
    do $r \leftarrow$ GET_ROUTE_IDENTIFIER(input)
        if EXISTS_IN_LIST(routes, $r$)
            then DROP_FROM_LIST(routes, $r$)
```

```
VALIDATE_EARNINGS(block, asks)
    proxy_asks $\leftarrow$ GET_PROXY_ASKS(asks, block)
    earnings $\leftarrow$ READ_EARNINGS_PER_OUTPUT(block)
    if GET_LENGTH(earnings) $\neq$ GET_LENGTH(proxy_asks)
        then ERROR(INVALID_COLLECTORS_NUMBER)
    for each ask in proxy_asks
    do VALIDATE_OUTPUT_EARNINGS(earnings, ask)
```

```
GET_PROXY_ASKS(asks, block)
 1  FORGET_PROXY_ASKS()
 2  for each ask in asks
 3  do SET_PROXY_ASK(ask)
 4  proxy_asks ← {}
 5  for each ask in asks
 6  do r ← GET_EARNER_IDENTIFIER(ask)
 7      if not INCLUDED_PROXY_ASK(r, proxy_asks) and
 8         NIL ≠ (proxy ← RECALL_PROXY_ASK(r))
 9         then ADD_TO_LIST(proxy_asks, proxy)
10  h ← READ_BLOCK_HEIGHT(block)
11  for each r in collectors
12  do k ← CREATE_ASK_IDENTIFIER(h, r)
13      ask ← CREATE_ASK(k, NIL, NIL, NIL, NIL)
14      WRITE_EARNINGS(ask, RECALL_ROUTE_EARNINGS(r))
15      ADD_TO_LIST(proxy_asks, ask)
16  return proxy_asks
```

```
SET_PROXY_ASK(ask)
 1  if 0 = ($f_b$ ← READ_EARNINGS(ask))
 2     then return
 3  r ← GET_EARNER_IDENTIFIER(ask)
 4  if NIL = (proxy ← RECALL_PROXY_ASK(r))
 5     then if NIL ≠ ($b_i$ ← RECALL_ROUTE_EARNINGS(r))
 6                then WRITE_EARNINGS(ask, $f_b + b_i$)
 7                         if EXISTS_IN_LIST(collectors, r)
 8                            then DROP_FROM_LIST(collectors, r)
 9              MEMORIZE_PROXY_ASK(r, ask)
10              return
11  WRITE_EARNINGS(proxy, $f_b$ + READ_EARNINGS(proxy))
```

```
GET_EARNER_IDENTIFIER(ask)
1  k ← READ_ASK_IDENTIFIER(ask)
2  return GET_ASKER(k)
```

```
INCLUDED_PROXY_ASK($r$, $proxy\_asks$)
 1  for each $ask$ in $proxy\_asks$
 2  do if GET_EARNER_IDENTIFIER($ask$) $= r$
 3         then return true
 4  return false
```

```
VALIDATE_OUTPUT_EARNINGS($earnings$, $ask$)
 1  $r \leftarrow$ GET_EARNER_IDENTIFIER($ask$)
 2  $f_b \leftarrow$ READ_EARNINGS($ask$)
 3  for each $output\_quota$ in $earnings$
 4  do if READ_EARNER_IDENTIFIER($output\_quota$) $= r$ and
 5         READ_OUTPUT_EARNINGS($output\_quota$) $= f_b$
 6         then return
 7  ERROR(INVALID_OUTPUT_EARNINGS)
```

```
VALIDATE_GIFT_RIGHTS($block$)
  1  $W_g \leftarrow 0$
  2  for each $k$ in $open\_asks$
  3  do $w_u \leftarrow$ RECALL_FAILING_TO_SELL($k$)
  4      $W_k \leftarrow$ RECALL_REVOKED_RIGHTS($k$)
  5      $l \leftarrow$ RECALL_CONCLUDED_LOSS(GET_ASKER($k$))
  6      if $w_u < (w_k \leftarrow ((2 \times l) - W_k))$
  7         then $w_k \leftarrow w_u$
  8      $W_g \leftarrow (W_g + (w_u - w_k))$
  9      VALIDATE_REVOKED_RIGHTS($k$, $w_k$, $block$)
 10  if READ_GIFT_RIGHTS($block$) $\neq W_g$
 11     then ERROR(INVALID_GIFT_RIGHTS)
```

```
VALIDATE_REVOKED_RIGHTS($k$, $w_k$, $block$)
 1  if $w_k = 0$
 2     then $w_k \leftarrow$ NIL
 3  if READ_REVOKED_RIGHTS($block$, $k$) $\neq w_k$
 4     then ERROR(INVALID_REVOKED_RIGHTS)
```

```
VALIDATE_CHAIN_PRUNING(block)
 1  SET_PRUNING_STATE(block)
 2  votes ← {}
 3  quorum_start ← $h_S$
 4  for each $X_b$ in voters
 5  do $b$ ← RETRIEVE_BLOCK($X_b$)
 6      ADD_PRUNING_VOTE(votes, $b$)
 7  if quorum ← (GET_LENGTH(votes) $\geqslant$ GET_QUORUM())
 8     then quorum_start ← GET_QUORUM_START(votes)
 9             if quorum_start $> h_S$ and validated
10                then $D_q$ ← CREATE_DUMMY_BLOCK(block)
11                else  if quorum_start $= h_S$
12                         then VALIDATE_HIGHEST_VOTER(block)
13                         else  ERROR(CONFLICTING_QUORUM_START)
14  VALIDATE_DUMMY_IDENTIFIER(block)
15  VALIDATE_PRUNING_VOTE(block)
```

```
SET_PRUNING_STATE($block$)
$D_q \leftarrow$ RETRIEVE_DUMMY_BLOCK()
$voters \leftarrow \{\}$
if $h_k < (highest\_start \leftarrow h_s)$
   then return
if $validated$ and
   $h_k <$ GET_BLOCK_HEIGHT(READ_HIGHEST_VOTER($D_q$))
   then return
$h \leftarrow$ READ_BLOCK_HEIGHT($block$)
$descending \leftarrow \{\}$
while $h_s \leqslant (h \leftarrow (h - 1))$
do $X_s \leftarrow$ GET_PARENT_IDENTIFIER($block$)
     $block \leftarrow$ RETRIEVE_BLOCK($X_s$)
     if $h > h_k$
        then ADD_TO_LIST($voters$, $X_s$)
        else if $h = h_k$
                 then $highest\_start \leftarrow$ GET_START($block$)
                         if $highest\_start = h_s$
                            then return
                         $X_c \leftarrow X_s$
                 else ADD_TO_LIST($descending$, $X_s$)
$routes \leftarrow$ GET_ROUTES(REVERT_LIST($descending$))
SET_HIGHEST_START($routes$, $descending$)
```

```
GET_START(block)
if $h_s > (height \leftarrow (\text{READ\_CHECKPOINT}(block) + 1))$
  then ERROR(CONFLICTING_CHECKPOINT_START)
if $h_s > (asks\_low \leftarrow \text{READ\_ASKS\_LOW}(block))$
  then ERROR(CONFLICTING_ASKS-LOW_START)
if $height = h_s$ or $asks\_low = h_s$
  then return $h_s$
$h \leftarrow \text{READ\_BLOCK\_HEIGHT}(block)$
$h_m \leftarrow (asks\_low - 1)$
$losses \leftarrow L_c \leftarrow 0$
while $h \geqslant asks\_low$ or $losses < L_c$
do if $h = h_s$
       then ERROR(CONFLICTING_MONEY-TARGET_START)
    $block \leftarrow \text{RETRIEVE\_PARENT}(block)$
    if $asks\_low > (h \leftarrow (h - 1))$
       then if $h = h_m$
                then $L_c \leftarrow \text{READ\_CURRENT\_LOSS}(block)$
            for each transaction in block
            do $z_t \leftarrow \text{GET\_SIZE\_IN\_BYTES}(transaction)$
               $losses \leftarrow (losses + z_t)$
if $h < height$
  then return $h$
return height
```

```
SET_HIGHEST_START(routes, blocks)
$h \leftarrow (h_k - 1)$
$descending \leftarrow \{\}$
$ascending \leftarrow \{\}$
$child \leftarrow$ NIL
for each $X_b$ in blocks
do $block \leftarrow \text{RETRIEVE\_BLOCK}(X_b)$
    ADD_TO_LIST(descending, $X_b$)
    SET_START(routes, block, child, descending, ascending, h)
    $ascending \leftarrow \text{REVERT\_LIST}(descending)$
    $child \leftarrow block$
```

```
SET_START($routes$, $block$, $child$, $descending$, $ascending$, $height$)
1  if $highest\_start \leqslant (h \leftarrow$ READ_BLOCK_HEIGHT($block$))
2     then return
3  for each $r$ in $routes$
4  do if ROUTE($r$, $block$, $child$, $descending$, $ascending$, $height$)
5        then $highest\_start \leftarrow h$
6             return
```

```
ROUTE($r$, $block$, $child$, $descending$, $ascending$, $height$)
 1  if CHAINER_OR_LESSEE($r$, $block$, $ascending$)
 2     then $h \leftarrow$ READ_BLOCK_HEIGHT($block$)
 3          $l_r \leftarrow$ GET_CHAINER_LOSS($block$)
 4          $W_r \leftarrow$ GET_REWARD($block$, $child$, $l_r$, true)
 5          return $((height - h) \times l_r) < W_r$
 6  if NIL $= (output \leftarrow$ GET_OUTPUT($r$, $block$))
 7     then return false
 8  $b_r \leftarrow$ GET_BALANCE($r$, $output$, $descending$)
 9  $l_r \leftarrow$ GET_ROUTE_LOSS($r$, $block$)
10  return ($b_r -$ GET_INACTIVITY_FEES($l_r$, $ascending$)) $> 0$
```

```
CHAINER_OR_LESSEE($r_p$, $block$, $ascending$)
1  $r \leftarrow$ READ_CHAINER_IDENTIFIER($block$)
2  if $r_p =$ GET_LAST_IDENTIFIER($r$, $block$, $ascending$)
3     then return true
4  if NIL $= (lease \leftarrow$ READ_LEASE($block$))
5     then return false
6  $r \leftarrow$ READ_LESSEE_IDENTIFIER($lease$)
7  return $r_p =$ GET_LAST_IDENTIFIER($r$, $block$, $ascending$)
```

```
GET_LAST_IDENTIFIER($r$, $block$, $ascending$)
1  $r \leftarrow$ GET_LAST_PROXY($r$, $block$)
2  for each $X_b$ in $ascending$
3  do $block \leftarrow$ RETRIEVE_BLOCK($X_b$)
4     $r \leftarrow$ GET_LAST_PROXY($r$, $block$)
5  return $r$
```

```
GET_BALANCE($r$, $output$, $descending$)
1  $b_e \leftarrow$ READ_EXPLICIT_BALANCE($output$)
2  for each $X_b$ in $descending$
3  do $block \leftarrow$ RETRIEVE_BLOCK($X_b$)
4     $earnings \leftarrow$ READ_EARNINGS_PER_OUTPUT($block$)
5     for each $output\_quota$ in $earnings$
6     do if READ_EARNER_IDENTIFIER($output\_quota$) $= r$
7           then $b_i \leftarrow$ READ_OUTPUT_EARNINGS($output\_quota$)
8                return $b_e + b_i$
9  return $b_e$
```

```
ADD_PRUNING_VOTE($votes$, $block$)
1  if NIL $= (X_b \leftarrow$ READ_VOTED_START($block$))
2     then if NIL $\neq$ ($lease \leftarrow$ READ_LEASE($block$))
3             then $X_b \leftarrow$ READ_PRE-VOTED_START($lease$)
4  if $X_b \neq$ NIL
5     then ADD_TO_LIST($votes$, GET_BLOCK_HEIGHT($X_b$))
```

```
GET_QUORUM()
1  return TRIM_TO_INTEGER(GET_LENGTH($voters$) $\div 2) + 1$
```

```
GET_QUORUM_START($votes$)
1  return TRIM_TO_INTEGER(GET_MEDIAN($votes$))
```

```
CREATE_DUMMY_BLOCK(block)
 1  D ← CREATE_EMPTY_DUMMY_BLOCK()
 2  b ← RETRIEVE_BLOCK($X_c$)
 3  while quorum_start $\leqslant$ (h ← READ_BLOCK_HEIGHT(b))
 4  do D ← SET_IMMUTABLE_BLOCK(D, b)
 5     b ← RETRIEVE_PARENT(b)
 6     if h = quorum_start
 7        then SET_DUMMY_DATA(D, b, block)
 8  N ← GET_HASH(GET_DUMMY_DATA(D))
 9  WRITE_DUMMY_HASH(D, N)
10  return D
```

```
SET_IMMUTABLE_BLOCK(D, block)
 1  $X_b$ ← GET_BLOCK_IDENTIFIER(block)
 2  $l_r$ ← GET_CHAINER_LOSS(block)
 3  if loss_height < quorum_start
 4     then WRITE_CHAINER_LOSS(D, $X_b$, $l_r$)
 5          WRITE_LOSS_HEIGHT(D, $X_b$, loss_height)
 6  WRITE_IMMUTABLE_BLOCK(D, $X_b$)
```

```
SET_DUMMY_DATA(D, b, block)
 1  $l_r$ ← GET_CHAINER_LOSS(b)
 2  WRITE_DUMMY_REWARD(D, GET_REWARD(b, NIL, $l_r$, true))
 3  WRITE_DUMMY_GIFT_RIGHTS(D, READ_GIFT_RIGHTS(b))
 4  WRITE_DUMMY_CURRENT_LOSS(D, READ_CURRENT_LOSS(b))
 5  WRITE_DUMMY_TIME(D, READ_BLOCK_TIME(b))
 6  WRITE_HIGHEST_VOTER(D, GET_PARENT_IDENTIFIER(block))
```

```
VALIDATE_HIGHEST_VOTER(block)
 1  $X_b$ ← GET_PARENT_IDENTIFIER(block)
 2  if READ_HIGHEST_VOTER($D_q$) $\neq X_b$
 3     then ERROR(CONFLICTING_HIGHEST_VOTER)
```

```
VALIDATE_DUMMY_IDENTIFIER(block)
$X_b \leftarrow$ READ_DUMMY_IDENTIFIER(block)
if not quorum
  then if $X_b \neq$ NIL
         then ERROR(ILLEGAL_DUMMY_REFERENCE)
       return
$N \leftarrow$ READ_DUMMY_HASH($D_q$)
if $X_b \neq$ CREATE_BLOCK_IDENTIFIER($N$, $quorum\_start - 1$)
  then ERROR(INVALID_DUMMY_IDENTIFIER)
```

```
VALIDATE_PRUNING_VOTE(block)
if NIL $=$ ($X_b \leftarrow$ GET_PRUNING_VOTE(block)) or
  (not validated and not quorum)
  then return
if NIL $=$ ($b \leftarrow$ RETRIEVE_BLOCK($X_b$))
  then ERROR(NONEXISTENT_VOTED_START)
$h \leftarrow$ READ_BLOCK_HEIGHT($b$)
if $h \leqslant quorum\_start$ or $h > highest\_start$
  then ERROR(ILLEGAL_VOTED_START)
```

```
GET_PRUNING_VOTE(block)
$X_b \leftarrow$ READ_VOTED_START(block)
if NIL $\neq$ (lease $\leftarrow$ READ_LEASE(block))
  then chainer_vote $\leftarrow$ ($X_b \neq$ NIL)
       $X_b \leftarrow$ READ_PRE-VOTED_START(lease)
       if $X_b \neq$ NIL and chainer_vote
         then ERROR(DOUBLE_PRUNING_VOTE)
return $X_b$
```

```
VALIDATE_CHAIN_STATE(block)
VALIDATE_GENERAL_MAX_HASH(block)
VALIDATE_ASKS_LOW(block)
VALIDATE_CHECKPOINT(block)
VALIDATE_MONEY_SUPPLY(block)
```

```
VALIDATE_GENERAL_MAX_HASH(block)
 1  $t_b \leftarrow$ READ_BLOCK_TIME(block)
 2  $t_p \leftarrow$ READ_BLOCK_TIME(RETRIEVE_PARENT(block))
 3  $i_b \leftarrow (t_b - t_p)$
 4  $d_w \leftarrow$ (READ_BLOCK_HEIGHT(block) $- h_w$)
 5  $N \leftarrow (x_g \times (1 + (((i_b \div I_b) - 1) \div d_w)))$
 6  if READ_GENERAL_MAX_HASH(block) $\neq N$
 7     then ERROR(INVALID_GENERAL_DIFFICULTY_TARGET)
```

```
VALIDATE_ASKS_LOW(block)
 1  $h \leftarrow low \leftarrow height \leftarrow$ READ_BLOCK_HEIGHT($b \leftarrow block$)
 2  while $h_w \leqslant (h \leftarrow (h - 1))$
 3  do $b \leftarrow$ RETRIEVE_PARENT($child \leftarrow b$)
 4      $l_r \leftarrow$ GET_CHAINER_LOSS($b$)
 5      $W_r \leftarrow$ GET_REWARD($b$, $child$, $l_r$, true)
 6      if $((height - h) \times l_r) < W_r$
 7         then $low \leftarrow h$
 8  if READ_ASKS_LOW(block) $\neq low$
 9     then ERROR(INVALID_ASKS_LOW)
```

```
VALIDATE_CHECKPOINT(block)
  1  $h \leftarrow$ READ_BLOCK_HEIGHT($b \leftarrow block$)
  2  $L_w \leftarrow 0$
  3  while $h \geqslant h_s$
  4  do $L_w \leftarrow (L_w +$ GET_CHAINER_LOSS($b$))
  5      if $h_s \leqslant (h \leftarrow (h - 1))$
  6         then $b \leftarrow$ RETRIEVE_PARENT($b$)
  7              $L_c \leftarrow$ READ_CURRENT_LOSS($b$)
  8         else $D \leftarrow$ RETRIEVE_DUMMY_BLOCK()
  9              $L_c \leftarrow$ READ_DUMMY_CURRENT_LOSS($D$)
 10      if $L_w \geqslant L_c$
 11         then if READ_CHECKPOINT(block) $\neq h$
 12                 then ERROR(INVALID_CHECKPOINT)
 13              return
 14  ERROR(MISSING_CHECKPOINT)
```

```
VALIDATE_MONEY_SUPPLY(block)
1  if READ_MONEY_SUPPLY(block) $\neq (M_c + M_v)$
2     then ERROR(INVALID_MONEY_SUPPLY)
```

```
UPDATE_CHAIN(block)
1  UPDATE_GLOBAL_STATE(block)
2  PERSIST_BLOCK(GET_BLOCK_IDENTIFIER(block), block)
3  if READ_BLOCK_HEIGHT(block) $\geqslant h_c$
4     then COMMIT()
```

```
UPDATE_GLOBAL_STATE(block)
1  UPDATE_CURRENT_HEIGHT(block)
2  UPDATE_STARTING_HEIGHT()
```

```
UPDATE_CURRENT_HEIGHT(block)
1  if READ_BLOCK_HEIGHT(block) $> h_c$
2     then PERSIST_CURRENT_HEIGHT($h_c + 1$)
```

```
UPDATE_STARTING_HEIGHT()
1  if not quorum
2     then return
3  if quorum_start $> h_s$
4     then if RETRIEVE_PRECEDING_START() $\neq h_s$
5              then UPDATE_PRECEDING_START()
6           PERSIST_DUMMY_BLOCK($D_q$)
7           PERSIST_STARTING_HEIGHT(quorum_start)
8           return
9  PERSIST_VALIDATED(true)
```

```
Update_Preceding_Start()
if NIL $\neq$ ($D \leftarrow$ Retrieve_Previous_Dummy())
  then Archive_Previous_Dummy($h_S$, $D$)
       $b \leftarrow$ Retrieve_Block($X_S$)
       while NIL $\neq$ ($b \leftarrow$ Retrieve_Parent($b$))
       do Prune_Block($b$, true)
$D \leftarrow$ Retrieve_Dummy_Block()
Persist_Previous_Dummy($D$)
Persist_Preceding_Start($h_S$)
```

```
Prune_Block(block, archive)
if archive
  then Archive_Block($h_S$, block)
$X_b \leftarrow$ Get_Block_Identifier(block)
Remove_Block($X_b$)
Remove_Proxy_Map($X_b$)
```

```
Validate_Branch(block)
if validated
  then Validate_Branch_Start(block)
Validate_Block(block)
while Read_Block_Height(block) $< h_C$
do if NIL = (block $\leftarrow$ Get_Candidate_Child(block))
      then Error(incomplete_branch)
   Validate_Block(block)
```

```
Validate_Branch_Start(block)
$D \leftarrow$ Retrieve_Dummy_Block()
$h \leftarrow$ Get_Block_Height(Read_Highest_Voter($D$))
if Read_Block_Height(block) $> h$
  then return
if not Restore_Previous_Dummy()
  then Error(branch_not_above_pruning_voters)
Validate_Branch_Start(block)
```

```
RESTORE_PREVIOUS_DUMMY()
 1  if NIL = ($start$ ← RETRIEVE_PRECEDING_START())
 2     then return false
 3  if $start < h_S$
 4     then $D$ ← RETRIEVE_PREVIOUS_DUMMY()
 5          PERSIST_DUMMY_BLOCK($D$)
 6          PERSIST_STARTING_HEIGHT($h_S \leftarrow start$)
 7          return true
 8  return false
```

```
GET_CANDIDATE_CHILD($block$)
 1  $X_b$ ← GET_BLOCK_IDENTIFIER($block$)
 2  if NIL = ($child$ ← GET_CHILD_FROM_PEERS($X_b$))
 3     then return NIL
 4  if not VALID_BLOCK_FORMAT($child$) or
 5     not VALID_BLOCK_HASH($child$) or
 6     $current\_time < (time$ ← READ_BLOCK_TIME($child$))
 7     then return NIL
 8  $h$ ← READ_BLOCK_HEIGHT($child$)
 9  $N$ ← READ_PARENT_HASH($child$)
10  if READ_BLOCK_HEIGHT($block$) = $(h - 1)$ and
11     READ_BLOCK_HASH($block$) = $N$ and
12     READ_BLOCK_TIME($block$) < $time$
13     then return $child$
14  return NIL
```

```
PRUNE_ORPHANED_BRANCHES()
  $h \leftarrow h_C$
  $pruned \leftarrow$ false
  $pruning \leftarrow$ true
  while $pruning$
  do $siblings \leftarrow$ RETRIEVE_BLOCKS_AT_HEIGHT($h \leftarrow (h-1)$)
      $pruning \leftarrow$ false
      for each $sibling$ in $siblings$
      do if not HAS_CHILDREN($sibling$)
            then PRUNE_BLOCK($sibling$, false)
                 $pruned \leftarrow pruning \leftarrow$ true
  if $pruned$
    then COMMIT()
```

```
HAS_CHILDREN($block$)
  $h \leftarrow$ READ_BLOCK_HEIGHT($block$)
  $siblings \leftarrow$ RETRIEVE_BLOCKS_AT_HEIGHT($h+1$)
  $N \leftarrow$ READ_BLOCK_HASH($block$)
  for each $sibling$ in $siblings$
  do if READ_PARENT_HASH($sibling$) $= N$
        then return true
  return false
```