

# State Fees 3

Pre-EIP proposal for discussion, version 3

# Copyright

Copyright and related rights waived via [CC0](#)

# Previous versions

This is the third published version of the proposal.

First version was published on 26th November 2018, and can be found [here](#).

Second version was published on 21st of January 2019, then underwent minor modifications on 24th and 27th of January 2019, and can be found [here](#).

# Main differences from the second proposal

- Replay protection for externally owned accounts changed from temporal to non-temporal to ensure that account nonces are never reused (reuse of nonces allow re-creation of contracts)
- Lock-ups are replaced with rent prepayments. Prepayments provide protection from dust grieving vulnerability, though temporary rather than permanent. Prepayments cannot be released, which avoids issues of changing economics of some smart contracts, like DEXs
- State counters are introduced to make the state size metrics trivially observable, as well as to provide future path for floating rent, if needed.

# Main differences from the second proposal

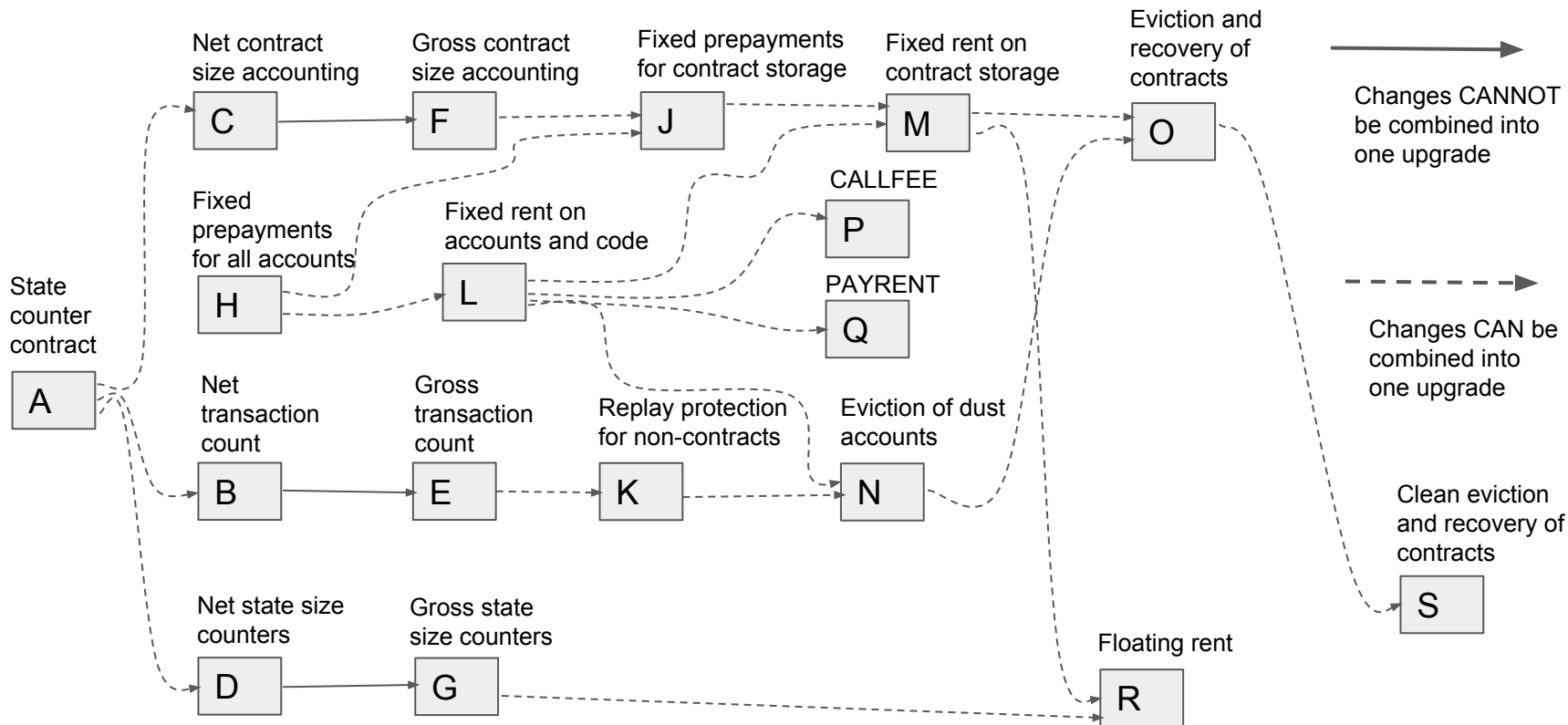
- Transaction format is not modified
- Functionally of `gaslimit` (field of a transaction) is extended so that `gaslimit*gasprice` limits prepayments
- Floating rent and “clean” eviction of contracts are re-added for completeness as optional changes

# Organisation of this proposal

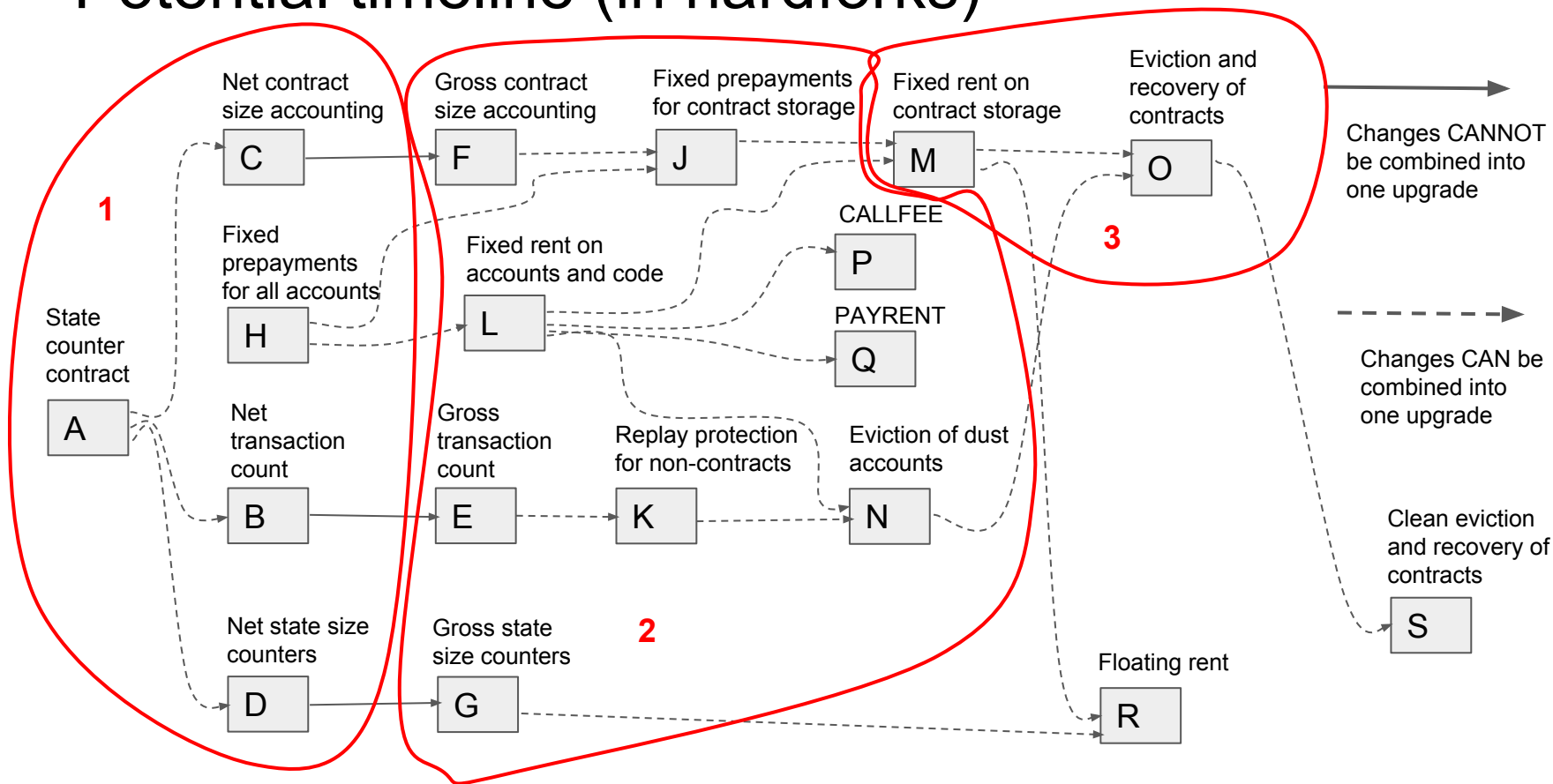
Proposal consists of the change cards, each given a capital letter, with the intention of each change potentially becoming an EIP (Ethereum Improvement Proposal). There are dependencies between the changes.

Diagram on the next page puts all proposed changes into a dependency graph. Solid arrows represent dependencies where the two changes cannot happen without the same protocol upgrade. Dash arrows represent dependencies where the two changes can be combined into the same upgrade.

# Dependencies between changes



# Potential timeline (in hardforks)





# Impact of changes

Change **H** (fork 1) - Gastoken becomes non-viable due to the increased cost of account creation.

Change **J** (fork 2) - Block gas limit can be increased without accelerating state size growth.

Change **N** (fork 2) - state size starts shrinking due dust account evictions

Change **O** (fork 3) - state size further shrinks due to contract evictions

# Change A - state counters contract

Prior to the block A, a contract is deployed with the following code: 0x60 0x20 0x60 0x00 0x80 0x80 0x35 0x54 0x90 0x52 0xF3, which corresponds to this assembly:

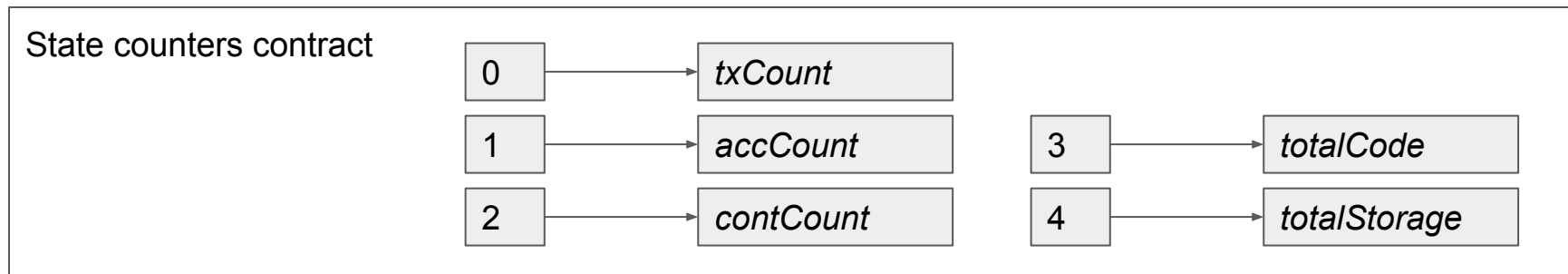
```
PUSH1 32; PUSH1 0; DUP1;      DUP1;      CALLDATALOAD; SLOAD;      SWAP1;      MSTORE;  
RETURN;
```

Call to this contract accepts one 32-byte argument, x, and returns the value of the storage item [x].

# Change A - state counters contract

Storage items of this contracts are populated by the ethereum client, and these items are used as state counters.

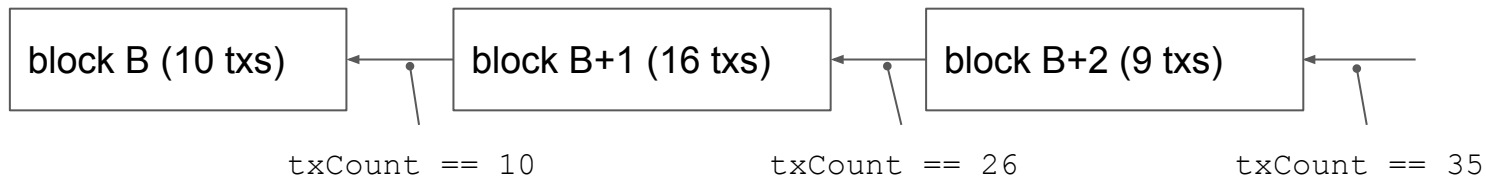
This approach of introducing state counters does not require changing the structure of the Patricia Merkle Tree and would be less disruptive for the infrastructure.



# Change B - net transaction count

A new field, with the location **0**, is added to the state counter contract. It will eventually contain *txCount*, the total number of transactions processed up until that point.

On an after block B, the field *txCount* is incremented after each transaction. Updating *txCount* means updating the storage of state counter contract at the location **0**. These changes are never reverted.



## Note on HUGE\_NUMBER

In the changes C and D, there is a constant `HUGE_NUMBER`. It needs to be large enough so that no real metrics (contract storage size, number of accounts, number of contracts, total size of code, total size of storage) will ever reach that number, and small enough that it fits in an unsigned 64-bit integer.

Current suggestion is to have `HUGE_NUMBER = 263`, which is binary representation of a single bit in a 64-bit number.

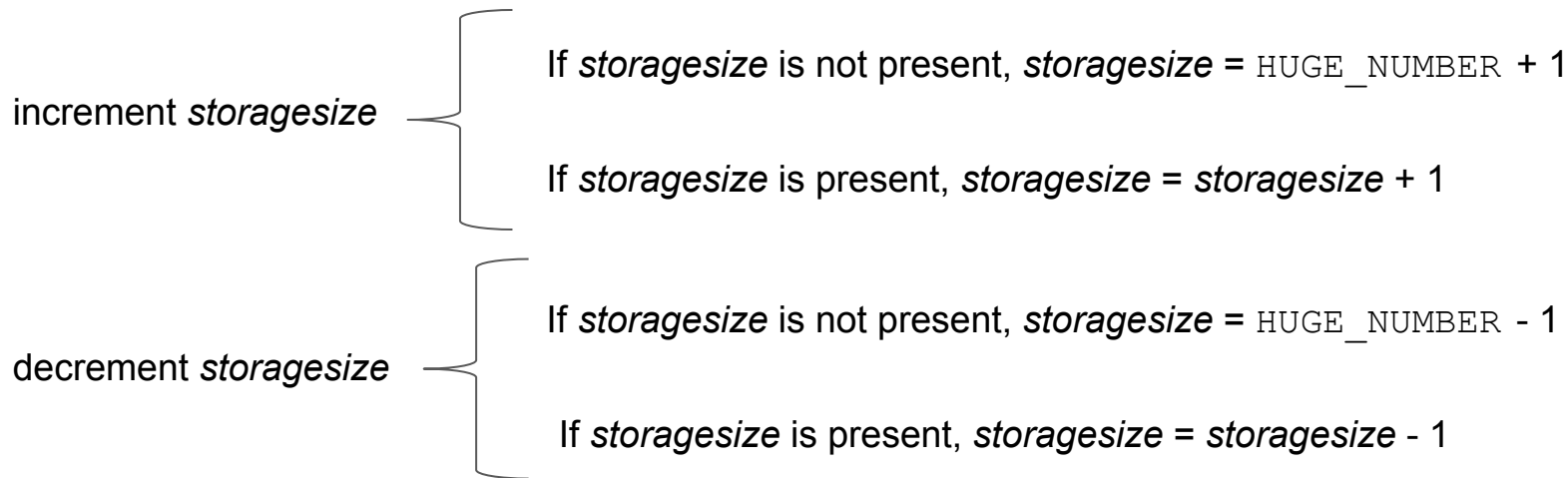
# Change C, net contract size accounting

Each contract gets a new `uint64` field, called *storagesize*. On and after block C, the semantics of the operation `SSTORE (index, value)` changes as follows:

- If previous value of the `[index]` is 0, and `value` is not 0, increment *storagesize* (semantics of “increment” described on the next page)
- If previous value of the `[index]` is not 0, and `value` is 0, decrement *storagesize* (semantics of “decrement” described on the next page)
- As with other state changes, changes of *storagesize* get reverted when the execution frame reverts, i.e. it needs to use the same techniques as storage values, like journalling (in Geth), and substates (in Parity).

Value of *storagesize* is not observable from contracts at this point.

# Change C, net contract size accounting



The idea is to make it decidable later whether the *storagesize* was ever incremented/decremented (presence of the field), and whether it has been converted from net to gross (by value being smaller than  $\text{HUGE\_NUMBER} - \text{<storage size at block C>}$ )

## Change D - net state size counters (*accCount*)

A new field, with the location 1, is added to the state counter contract. It will eventually contain *accCount*, the total number of non-contract accounts present in the state.

At the beginning of block D, *accCount* is set to `HUGE_NUMBER`.

On an after block D, any EVM action (transaction or `CALL` with `value!=0` to a “fresh” address) that leads to creation of a new non-contract account, increments *accCount*, and any EVM action (none yet) that leads to removal or replacement of a non-contract account, decrements *accCount*. These increments and decrements are state changes, and they get reverted when execution frame reverts.



## Change D - net state size counters (*contCount* & *totalCode*)

A new field, with the location 2, is added to the state counter contract. It will eventually contain *contCount*, the total number of contract accounts present in the state.

A new field, with the location 3, is added to the state counter contract. It will eventually contain *totalCode*, the aggregate size of bytecode for all contract accounts present in the state.

At the beginning of block D, *contCount* is set to `HUGE_NUMBER`, and *totalCode* is sent to `HUGE_NUMBER`

## Change D - net state size counters (*contCount* & *totalCode*)

On and after block D, any EVM action (transaction to “void” address, `CREATE`, `CREATE2`) that leads to creation of a new contract account, increments *contCount*, and increases *codeSize* by the size of the bytecode of the new contract.

On and after block D, any EVM action (`SELFDESTRUCT`) that leads to removal of a non-contract account, decrements *accCount*, and decreases *codeSize* by the size of the bytecode of the removed contract.

These increments, decrements, increases and decreases are state changes, and they get reverted when execution frame reverts.

## Change D - net state size counters (*totalStorage*)

A new field, with the location 4, is added to the state counter contract. It will eventually contain *totalStorage*, the aggregate number storage items for all contract accounts present in the state.

At the beginning of block D, *totalStorage* is set to `HUGE_NUMBER`.

## Change D - net state size counters (*totalStorage*)

On and after block D, the semantics of the operation `SSTORE (index, value)` changes as follows:

- If previous value of the `[index]` is 0, and `value` is not 0, increment *totalStorage*
- If previous value of the `[index]` is not 0, and `value` is 0, decrement *totalStorage*
- As with other state changes, changes of *totalStorage* get reverted when the execution frame reverts

## Change E - gross transaction count

At the beginning of block E, field *txCount* of the state counter contract, is increased by the total number of transaction that happened in the network since the genesis block and before the block B.

# Change F - gross contract size accounting

Client software is shipped with a mapping, for all contracts existing at the point after block C-1 and before block C, of contract addresses to the actual number of items. We denote the storage size of a contract at block C as `<size_at_C>`

On and after block D, any modification of a contract apply this check:

- If *storagesize* field is not present, *storagesize* = `<size_at_C>`
- If *storagesize* field is present, and its value is less than `HUGE_NUMBER-<size_at_C>`, leave *storagesize* unchanged
- If *storagesize* field is present, and its value is more than `HUGE_NUMBER-<size_at_C>`, decrease it by `HUGE_NUMBER-<size_at_C>`

# Change F - gross contract size accounting

After block D, the value of *storagesize* is observable by these rules:

- If *storagesize* field is not present, observed value is `<size_at_C>`
- If *storagesize* field is present, and its value is less than `HUGE_NUMBER-<size_at_C>`, observed value is *storagesize*
- If *storagesize* field is present, and its value is more than `HUGE_NUMBER-<size_at_C>`, observed value is *storagesize*-`HUGE_NUMBER-<size_at_C>`

# Change G - gross state size counters

At the beginning of block G, fields *accCount*, *contCount*, *totalCode*, and *totalStorage* are all increased by the respective number of non-contract accounts, number of contract accounts, aggregate size of the bytecode of all contracts in the state, and aggregate number of storage items in all contracts in the state, all taken as of the state between blocks D-1 and D. Right after that, these 4 fields are decreased by `HUGE_NUMBER`.



# Prepayments explained

Prepayments is a mechanism that has two main functions:

1. Providing temporary protection from dust griefing vulnerability to the existing contracts. Because any state expansion requires prepayment of rent, the effects of potential dust griefing attacks are delayed by the time determined by `PREPAYMENT/RENT_PER_BLOCK` (in blocks)
2. Making state expansion more costly in a way that cannot be circumvented by cooperation with miners.
3. Requires definition of two constants (that can be changed via forks):  
`ACCOUNT_PREPAYMENT` **and** `STORAGE_PREPAYMENT`

# Safety of prepayments

After prepayments are introduced, there can be two reasons for ether to be deducted from tx.origin: purchasing and spending gas, and spending gas for prepayments. Gaslimit of a transaction currently plays a role of safety limit, where  $\text{gaslimit} * \text{gasprice}$  represents the maximum amount of wei the sender (tx.origin) authorises the transaction to deduct from its account.

After prepayments are introduced,  $\text{gaslimit} * \text{gasprice}$  will still represent the maximum amount of wei spend, but it will be used for both gas purchases and prepayments, as necessary.

# Change H - fixed prepayments for all accounts

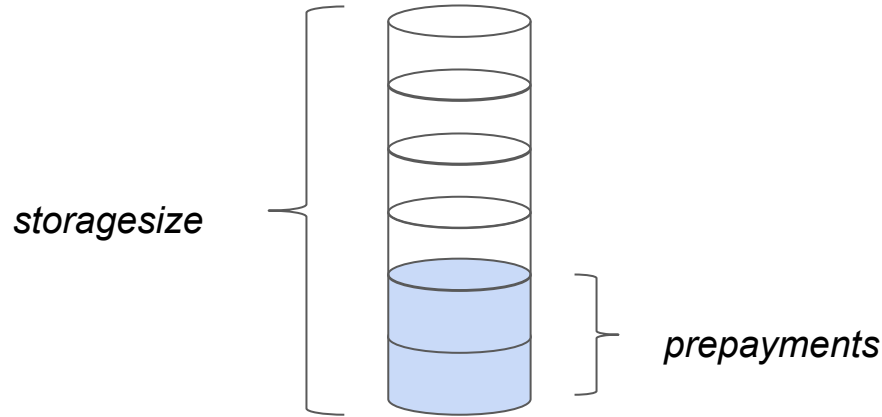
On and after block H, every newly created account gets a new field *rentbalance*, which can be negative.

On and after block H, any operation that leads to the creation of a new account, deducts the amount `ACCOUNT_PREPAYMENT` from `tx.origin`. This amount is added to the *rentbalance* field of the created account.

On and after block H, any operation that modifies an account that does not yet have *rentbalance* field, deducts the amount `ACCOUNT_PREPAYMENT` from `tx.origin`. This amount is added to the *rentbalance* field of the modified account. This is a anti-hoarding measure.

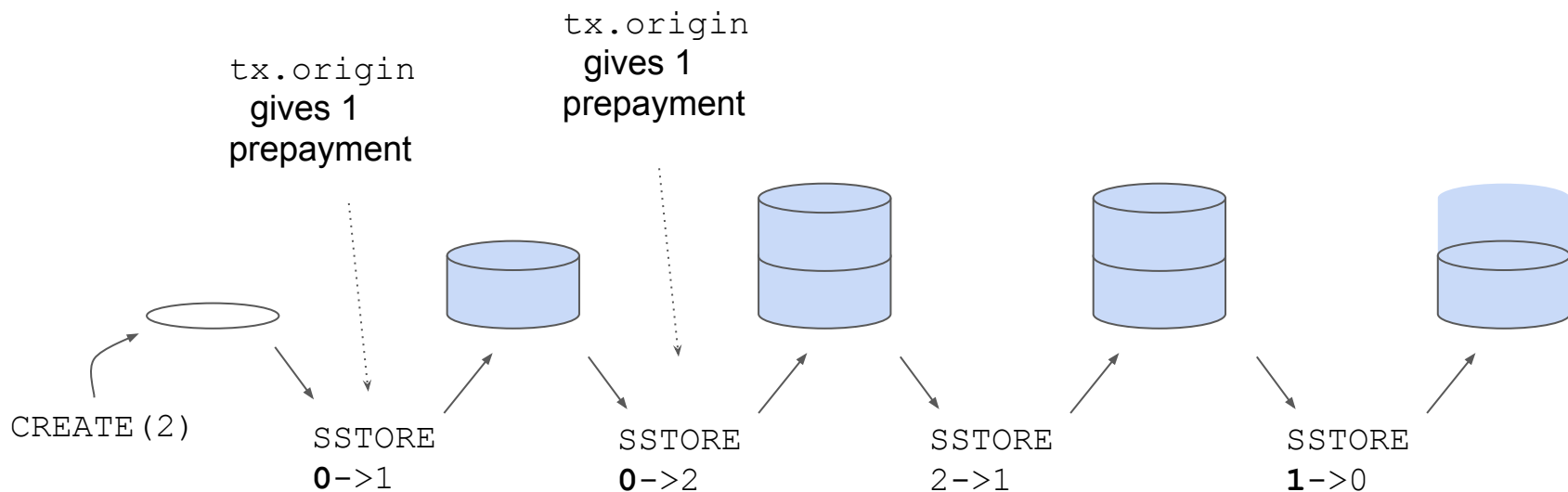
# Change J - fixed prepayments for contract storage

Analogy with the glass (contract). Capacity of the glass corresponds to the storage size of the contract. Amount of water in the glass corresponds to the prepaid amount.



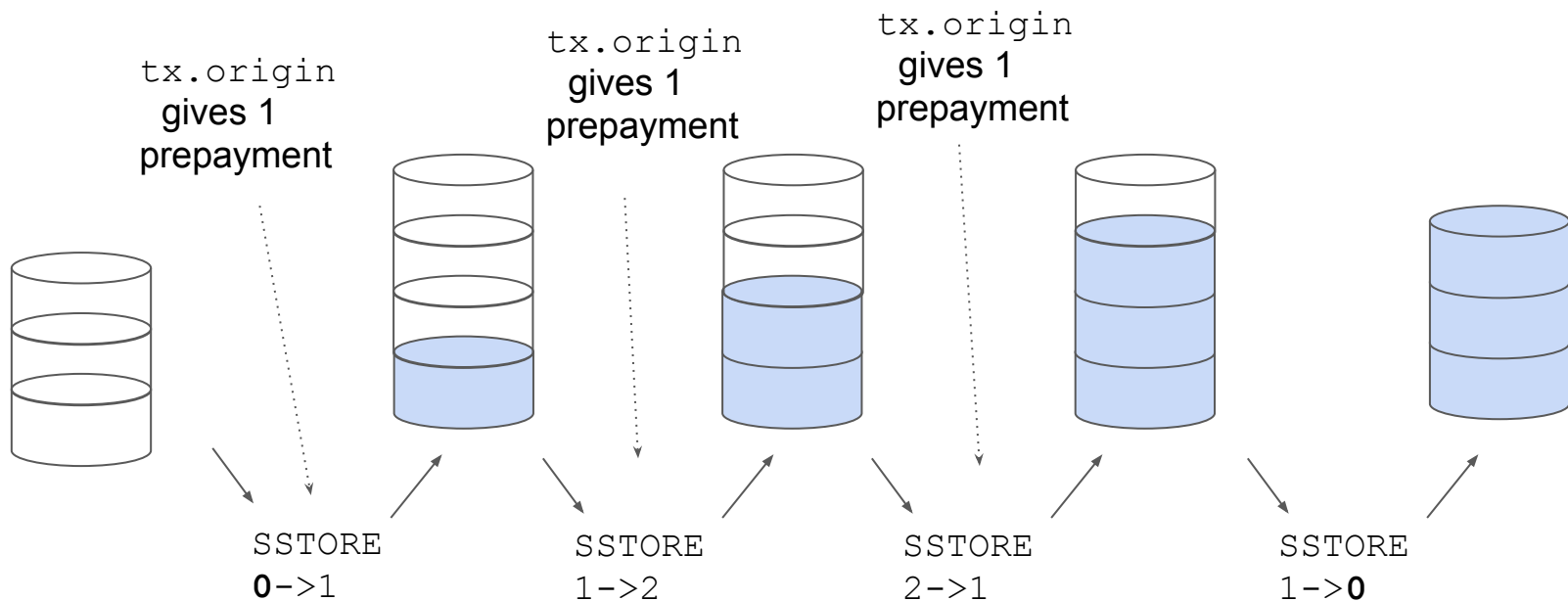
# Change J - fixed prepayments for contract storage

On and after block J, for newly created contracts, their entire storage needs to be “filled” with the prepayments:



# Change J - fixed prepayments for contract storage

On and after block J, for pre-existing contracts, the storage needs to be filled with the prepayments before any release can happen



# Change J - fixed prepayments for contract storage

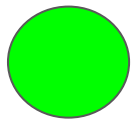
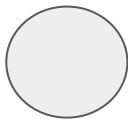
On and after block J, semantics of operation `SSTORE (index, new)` changes as shown below. Value of `[index]` at the beginning of transaction (not a single execution frame) is `original`. Value of `[index]` prior to the execution of `SSTORE` is `current`.

Without loss of generality, we will analyse only three possible values of `original`, `current`, and `new`: **0**, **1**, and **2**. “**0**” represents zero (value that can be removed from the state), “**1**” and “**2**” represent two distinct non-zero values (it does not matter what they are, all that matters is that they are non-zero and not equal to each other).

# Change J - fixed prepayments for contract storage

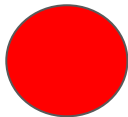
We will present the semantic change of `SSTORE` as transitions between 4 possible states:

Ground state - no  
change in the state of the  
contract storage



New storage item is allocated  
(0 -> 1 or 2)

Storage item has been  
removed (1 or 2 -> 0)



Storage item changed its value  
(1 -> 2 or 2 -> 1)



# Change J - fixed prepayments for contract storage

It is quite straightforward to map all possible values of triple (`original`, `current`, `new`) to corresponding transition between the four states. Source state of transition depends only on the pair (`original`, `current`):

original -> v-current	0	1	2
0	ground 	removed 	removed 
1	new 	ground 	changed 
2	new 	changed 	ground 

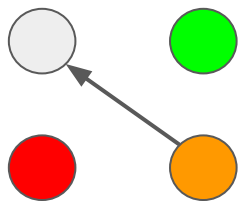
# Change J - fixed prepayments for contract storage

Destination of transition depends only on the pair (`original`, `new`), in an identical way:

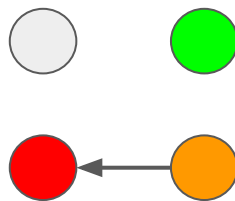
<code>original -&gt;</code> <code>v-new</code>	<b>0</b>	<b>1</b>	<b>2</b>
<b>0</b>	ground 	removed 	removed 
<b>1</b>	new 	ground 	changed 
<b>2</b>	new 	changed 	ground 

# Change J - fixed prepayments for contract storage

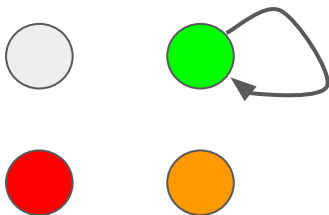
For example, the triple  
(original=1, current=2, new=1)  
represents transition:



Triple (original=1, current=2,  
new=0) represents transition:



Triple (original=0, current=1, new=2)  
represents a no-op transition:



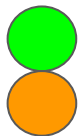
# Change J - fixed prepayments for contract storage

Translation of a transition between 4 states and the actual semantics of `SSTORE` requires defining that each of the 4 states mean. Proposed meaning includes two cases:

1. Contract pre-existed before introduction of prepayments and contains some storage that does not have corresponding prepayments, i.e.  $rentbalance < storagesize * STORAGE\_PREPAYMENT$
2. Contract either created after introduction of lockups, or already “caught up” with the lock ups, i.e.  $rentbalance \geq storagesize * STORAGE\_PREPAYMENT$

# Change J - fixed prepayments for contract storage

*rentbalance* < *storagesize*\*STORAGE\_PREPAYMENT



*balance*(tx.origin) -= STORAGE\_PREPAYMENT  
*rentbalance* += STORAGE\_PREPAYMENT



no-op

*rentbalance* >= *storagesize*\*STORAGE\_PREPAYMENT



*balance*(tx.origin) -= STORAGE\_PREPAYMENT  
*rentbalance* += STORAGE\_PREPAYMENT

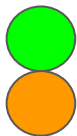


no-op

# Change J - fixed prepayments for contract storage

If we combine translation with the state transitions, this is what we would get in our previous examples:

*rentbalance* < *storage*size\*STORAGE\_PREPAYMENT



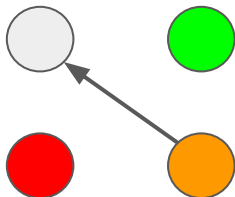
*balance*(tx.origin) -= STORAGE\_PREPAYMENT

*rentbalance* += STORAGE\_PREPAYMENT



no-op

(original=1,  
current=2, new=1)



Semantics:  
no-op

# Change J - fixed prepayments for contract storage

Prepayments cannot be charged and released via “piggybacking” on the gas model. This is because, when transaction reverts, gas that is already spent, is not refunded, but prepayments are. Therefore, it is easier to treat prepayments as changes in the balance of `tx.origin`. Obviously, prior to the reducing the balance, there needs to be a check to see if there are enough funds in the account. As mentioned before, the invariant

$$\text{gasUsed} * \text{gasPrice} + \text{prepayments} \leq \text{gasLimit} * \text{gasPrice}$$

holds, and `OutOfGas` exception is thrown if an operation would violate it.

# Change J - fixed prepayments for contract storage

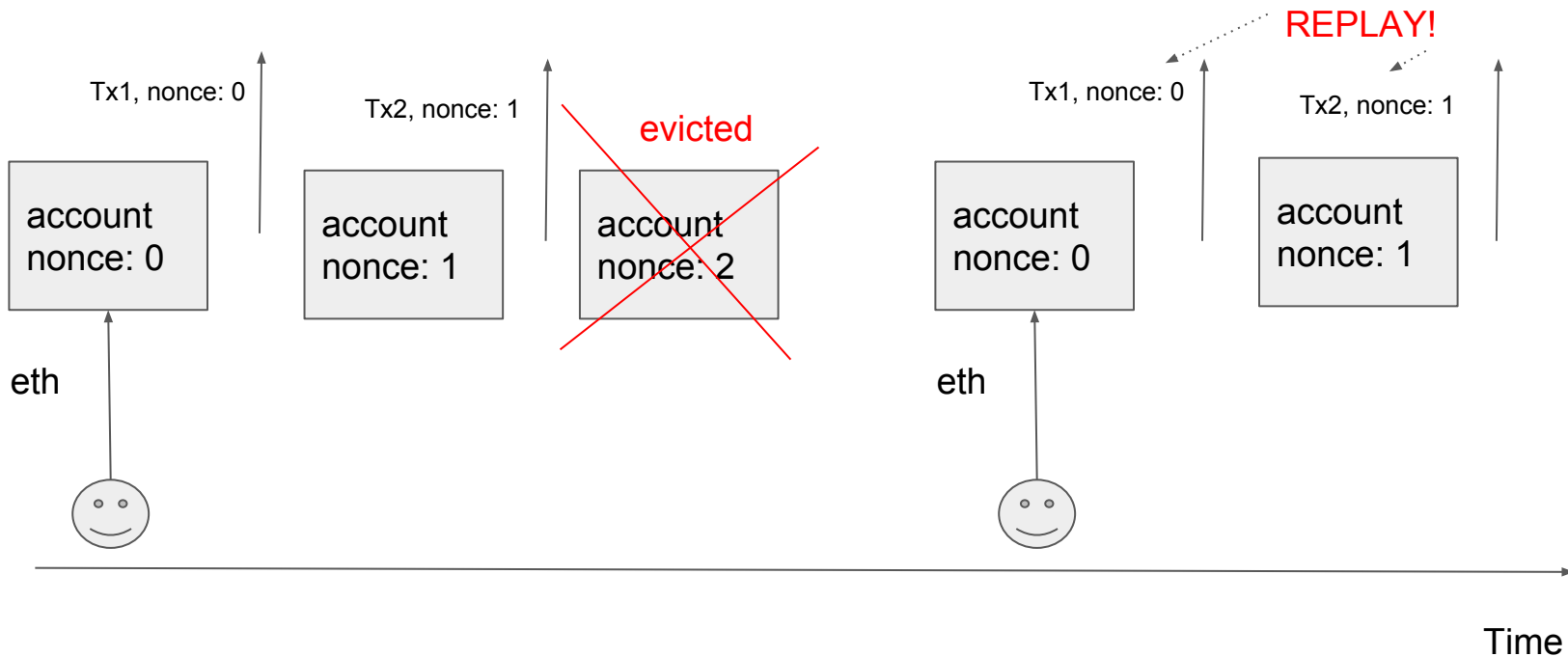
How big the prepayments should be?

It can be roughly calculated given the “target storage size” (this can be discovered using simulations) and the “target prepaid eth”. For example, if target storage size of 500 million items, and we want 10 million ETH to be prepaid to fill it up, the lock up price would be 0.02 ETH per item.

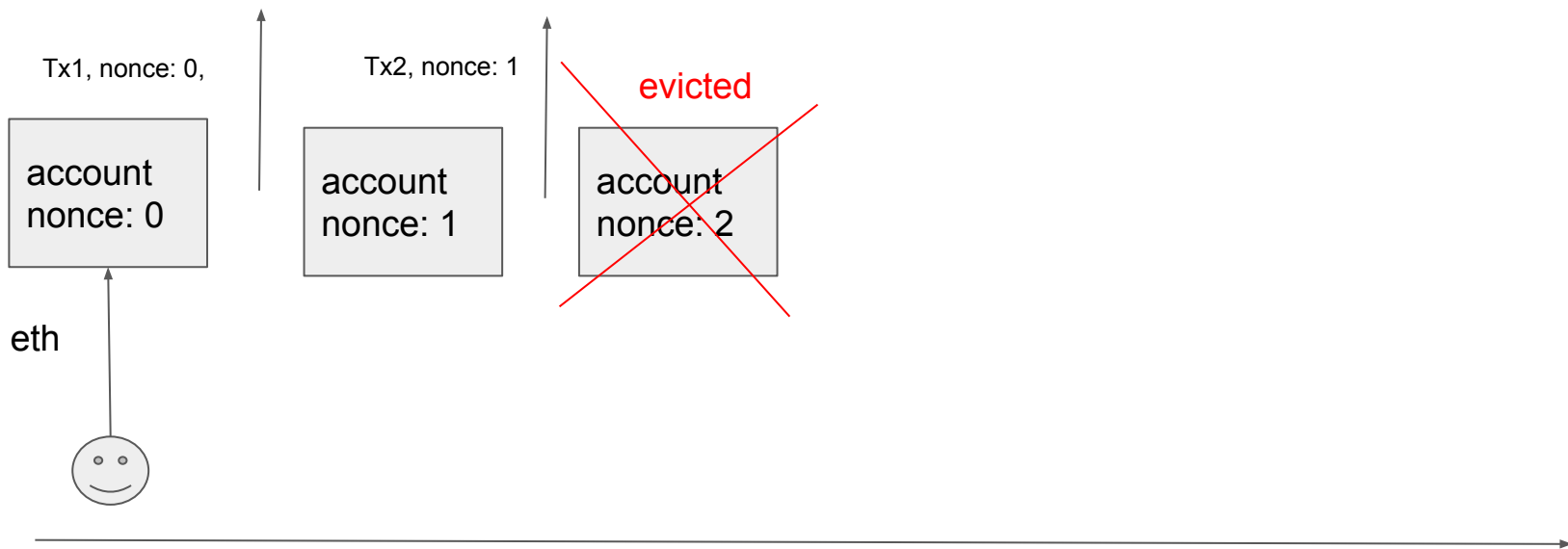


# Need for replay protection for new accounts

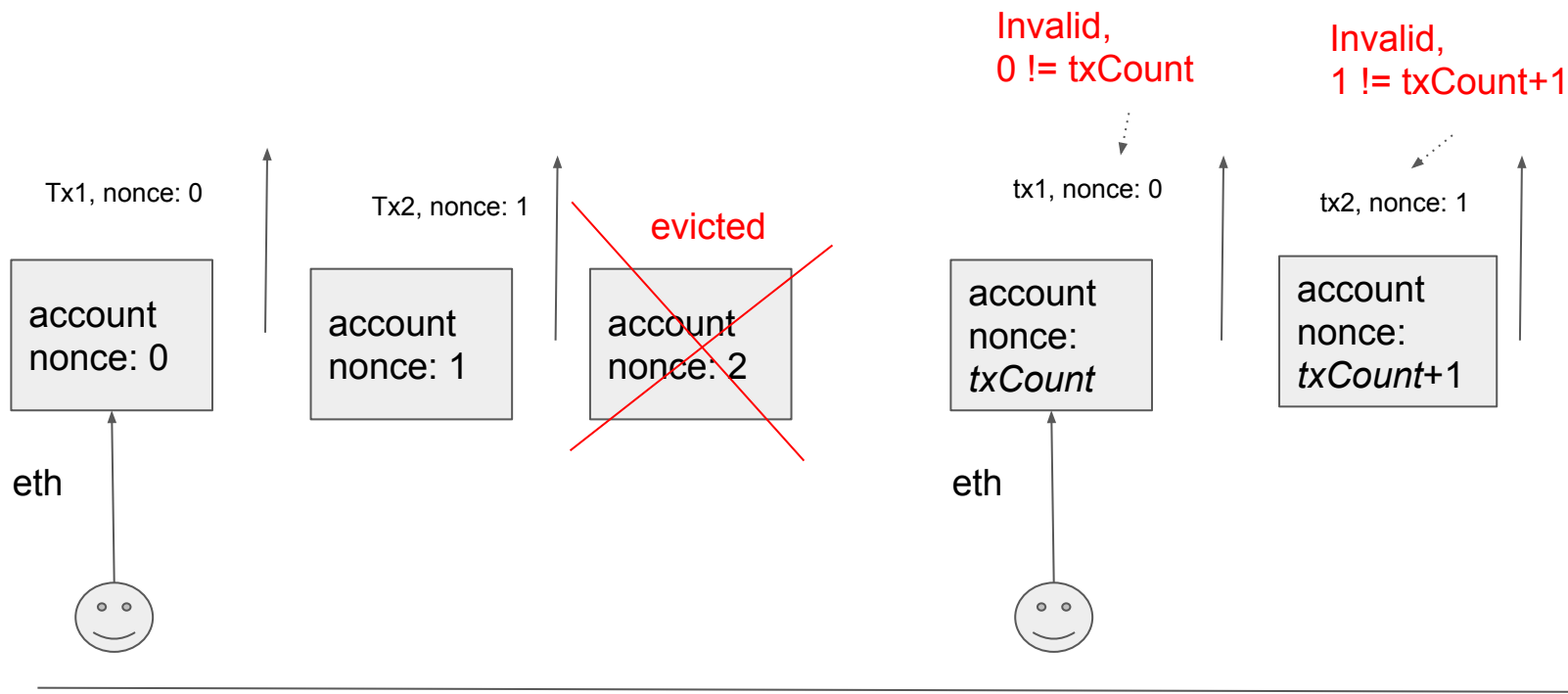
Eviction creates a replay problem when account is recreated



# Change K, replay protection for non-contract accounts



# Change K, replay protection for non-contract accounts



# Change K - replay protection for non-contract accounts

On and after block K, whenever a non-contract account is created (by transaction or `CALL` with value  $\neq 0$  to a “fresh” address), the nonce of the created account is set to the value of the *txCount* state counter (item with location 0 in the state counter contract).

# Change L - fixed rent on accounts and code

On and after block L, every account gets an extra field: *rentblock* (last time rent was calculated), with default = block L

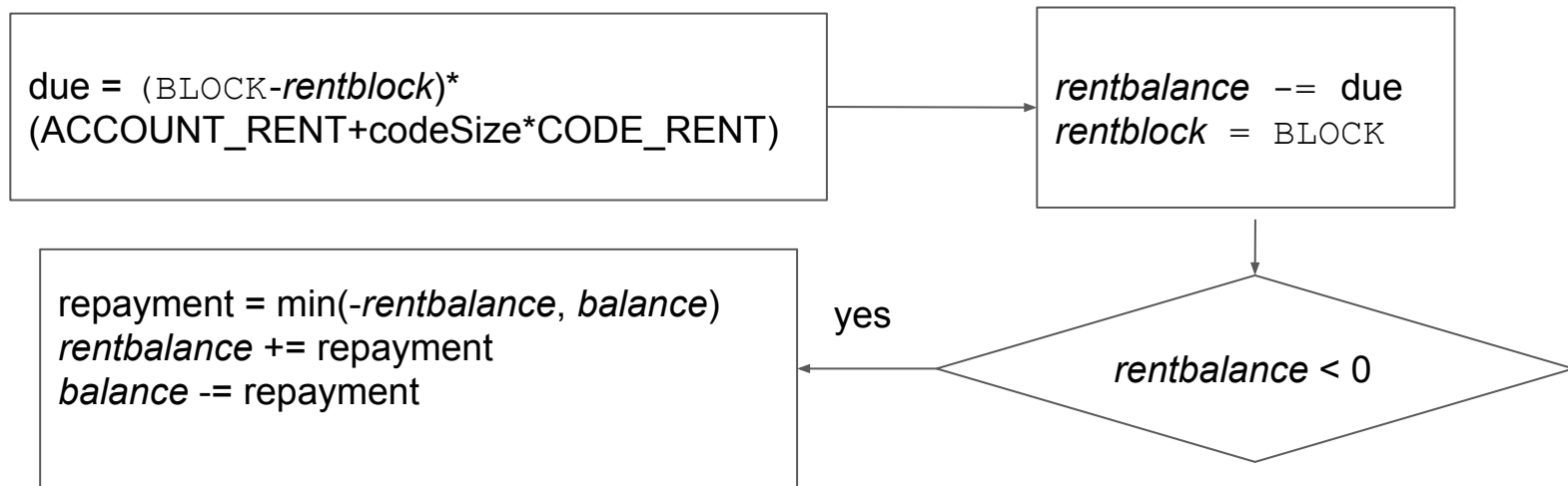
Two rent constants are defined:

1. ACCOUNT\_RENT - rent charge in wei per account per block, for example,  $2 \cdot 10^9$ .
2. CODE\_RENT - rent charge in wei per byte of code per block, for example,  $1 \cdot 10^7$

Pre-compiled contracts are exempt from rent.

# Change L - fixed rent on accounts and code

Calculation of dues happens whenever account is **modified** by EVM. Both *rentbalance* and *balance* fields of an account may be updated as a result.



# Change L - fixed rent on accounts and code

Calculation of rent happens during the Block Finalisation. Currently, the Yellow Paper describes 4 stages of Block Finalisation:

- 1) Validate omers
- 2) Validate transactions
- 3) Apply rewards
- 4) Verify state and block nonce

Rent calculation happens before stage (4), because it modifies state, but after stage (3), because stage (3) can increase the set of modified accounts.

# Change M - fixed rent on contract storage

A new rent constant is introduced:

STORAGE\_RENT - amount of wei charged for 1 item of storage without lock-ups, per block



# Change M - fixed rent on contract storage

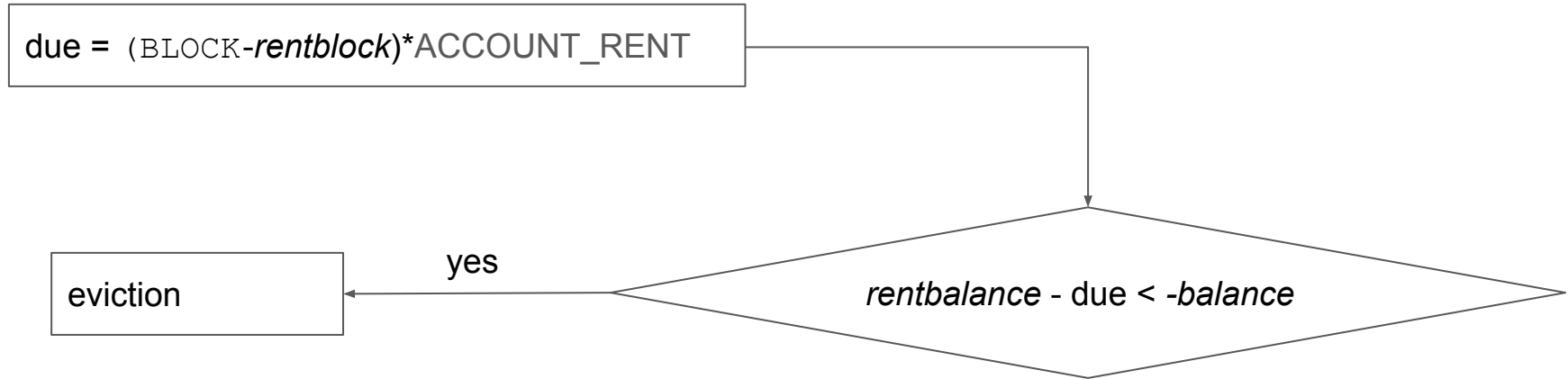
Rent calculation for contracts is modified, with the difference of how due is calculated:

$$\text{due} = (\text{BLOCK\_rentblock}) * ( \\ \text{STORAGE\_RENT} * \text{storageSize at the start of the block} + \\ \text{ACCOUNT\_RENT} + \\ \text{CODE\_RENT} * \text{codeSize at the start of the block} )$$

Values of *storageSize*, and *codeSize* are taken as values at the beginning of the current block, to avoid being charged more for all the blocks since the last modification. Value of *storageSize* is taken as “observable” value described in Change F

# Change N - Eviction of dust accounts

On and after block N, eviction check is performed at the end of a transaction for non-contract accounts that were touched during that transaction. If account is not evicted, eviction check does not change the value of the account.



# Change N - Eviction of dust accounts

Non-contract accounts are defined as accounts with the codeHash == keccak256(nil), which is `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`

On and after block N, when an account is evicted, the state size counter *accCount* is decremented.

# Change O - Eviction and recovery of contracts

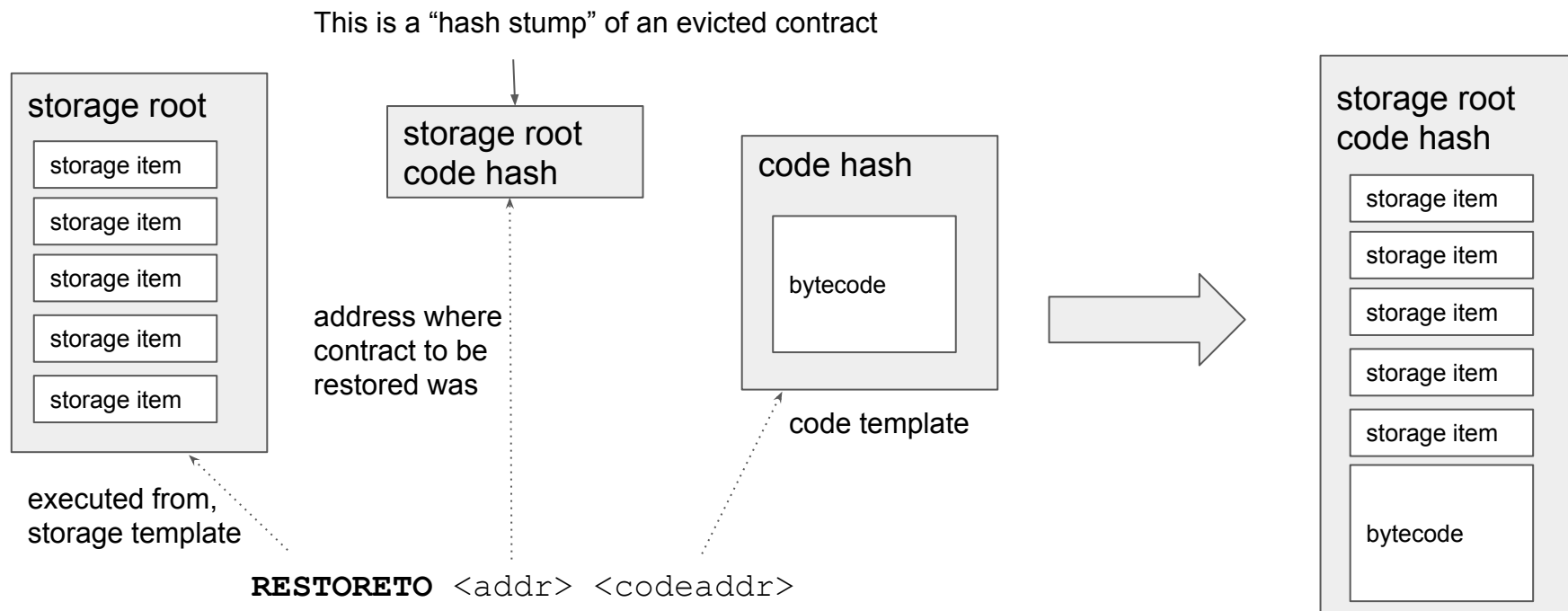
Eviction of contracts happens upon “touching” them in a transaction. Unlike the eviction of a non-contract account, eviction of a contract leaves a “hash stub” in the state. This hash stub can be later used to reconstruct the contract if it is still needed.

Once contract becomes a “hash stub”, it is treated as non-existent by any operations except `RESTORETO`, described as a diagram later, and a draft EIP here: <https://github.com/ethereum/EIPs/pull/1682>

TODO: Look at how to distinguish RLP of hash stub from RLP of a contract

# Change O - Eviction and recovery of contracts

`RESTORETO` opcode allows restoration from a “hash stump” left in the state.



# Change O - Eviction and recovery of contracts

On and after block O, when a contract gets evicted, *contCount* state size counter is decremented, and the state size counter *totalCode* counter is decreased by the size of the evicted contract's bytecode. State size counter *totalStorage* is decreased by the number of storage items in the evicted contract.

On and after block O, when a contract is restored, *contCount* state size counter is incremented, and the state size counter *totalCode* is increased by the size of the restored contract's bytecode. State size counter *totalStorage* is increased by the number of storage items in the restored contract.

# Change P - gathering rent for the code


Since contracts need to pay rent for maintenance of their account, and their code, the free-riders problem does not completely go away. In order keep totally unmanned contracts viable and also promote code reuse, contracts have an additional parameter: *callfee*. When set (most probably during deployment), each invocation of the contract (via `CALL`, `CALLCODE`, `DELEGATECALL`, or `STATICCALL`) is charged extra fee equal to *callfee*. This is added to the contract's *rentbalance*, and cannot be turned back to ETH.

Setting *callfee* is done via a new opcode `CALLFEE`.


# Change Q - New opcode PAYRENT

New opcode `PAYRENT` is introduced to top up *rentbalance* by spending ETH, and `RENTBALANCE` (to read *rentbalance*). This can help keep existing contracts alive until they are migrated.

`PAYRENT <target_contract> <amount>`



Contract whose  
*rentbalance* will be topped  
up



Top-up amount



# Change R - floating rent and prepayments

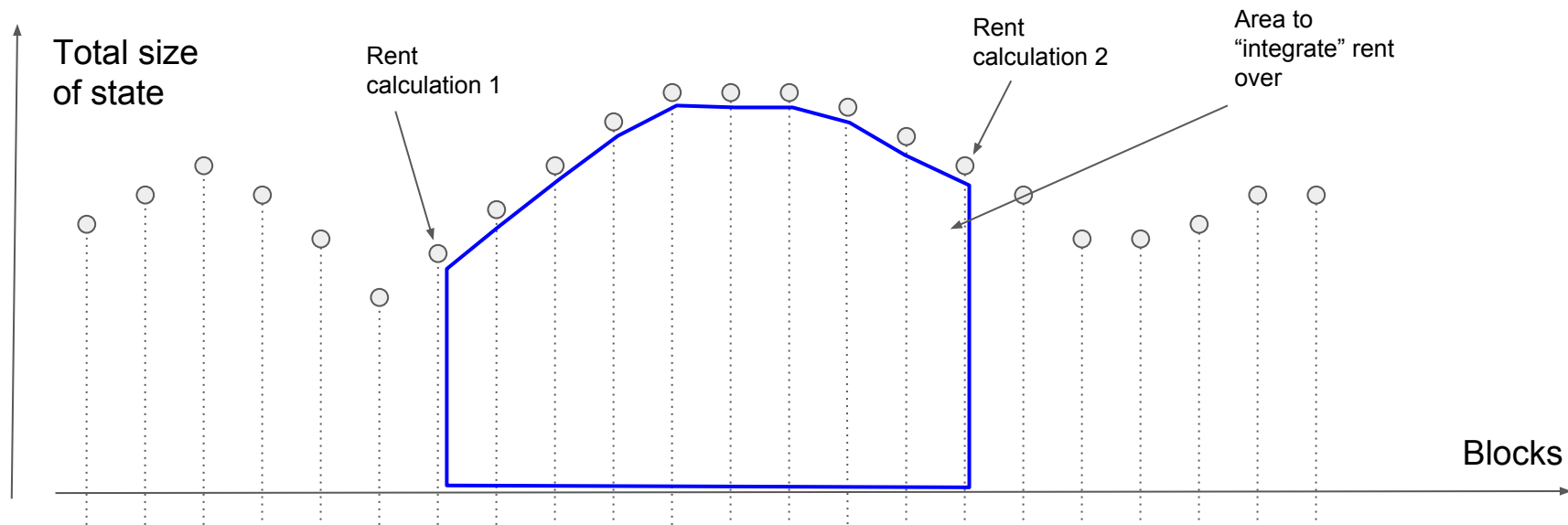
The idea is to introduce of storage upper limit and floating rent price, depending on the “state size pressure”. State size pressure is the measure of how closely the current size of the state is to the upper limit.

Some more info on maintenance fees:

<https://gist.github.com/zsfelfoldi/c40ff6637b9a6a095ddada87eb0d4891>

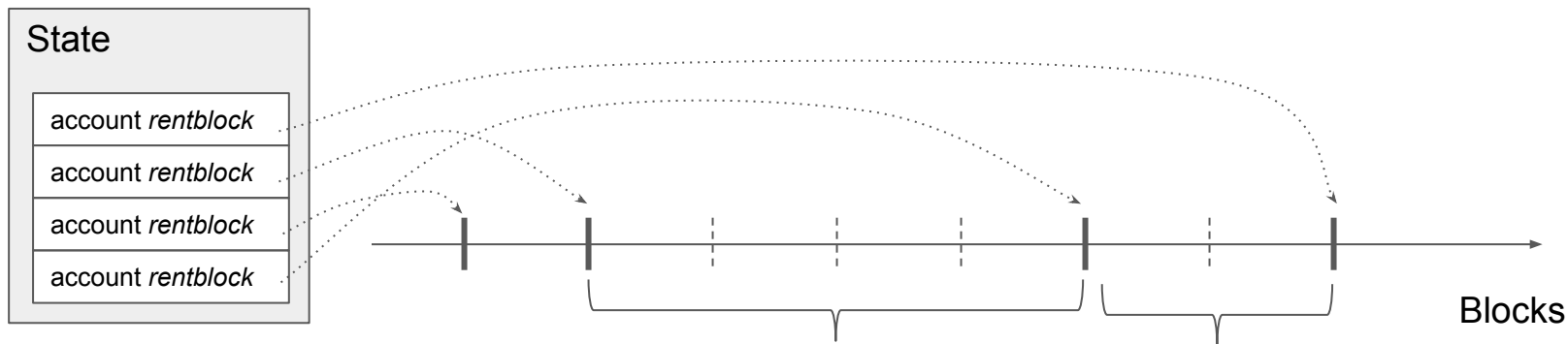
# Change R - floating rent and prepayments

Since rent is calculated at each modification of an account, the *storagesize* field does not change in between, so the only complexity is to aggregate the potentially changing rent over the calculation period.



# Change R - floating rent and prepayments

In order to integrate over the history of state size, the protocol needs to keep track of that history for  $\text{BLOCK} - \min(\text{rentblock})$  blocks, where “min” is taken over all the accounts that are presently in the state. Alternatively, it might be more space efficient to only keep track of intervals between various values of *rentblock*



only keep aggregated values for the intervals

# Change S - clean eviction and recovery of contracts

2 options so far:

- 1) Vitalik's suggested exclusion proofs, which imposes minimum live time on contracts, chapters 8-9 of <https://github.com/ethereum/research/blob/master/papers/pricing/ethpricing.pdf>
- 2) Graveyard tree - state includes a merkle root of the tree containing all removed contracts. To remove a contract, one needs to show the path in the graveyard tree where the contract's hash will live (this requires knowledge of the history of all removals). To resurrect a contract, one needs to show the path to the contract in the graveyard tree. The root of the tree gets updated that the contract is not in the tree anymore (it is now alive).