State Rent 2

Pre-EIP proposal for discussion, version 2

Copyright

Changes and argumentation

This proposal represents a reworking of the first proposal.

Based on:

1. Constructive feedback received for the 1st proposal since its publication on 26 Nov 2018, including this
2. Framework for change plans (it is still Work In Progress)
3. Alternative Fund Lock proposal
4. Notes on Proof Of Concept: https://www.symphonious.net/2019/01/14/ethereum-state-rent-proof-of-concept/
5. Other discussions

Main differences from the first proposal

- Linear Cross-Contract Storage removed, since it is possible to emulate its functionality by contracts using CREATE2 opcode.
- Priority Queue for eviction removed, eviction would be based on “touching”, under the presumption that miners would not be able to censor evictions (if they had motivation to do so).
- Calculation of the correct contract storage size (accounting) introduced before the rent, to avoid some edge cases, where rent could become negative.
- Lock-ups with a fixed price introduced, to prevent dust griefing attacks on pre-existing contracts, and to reduce the need to rewrite most of contracts.

Main differences from the first proposal

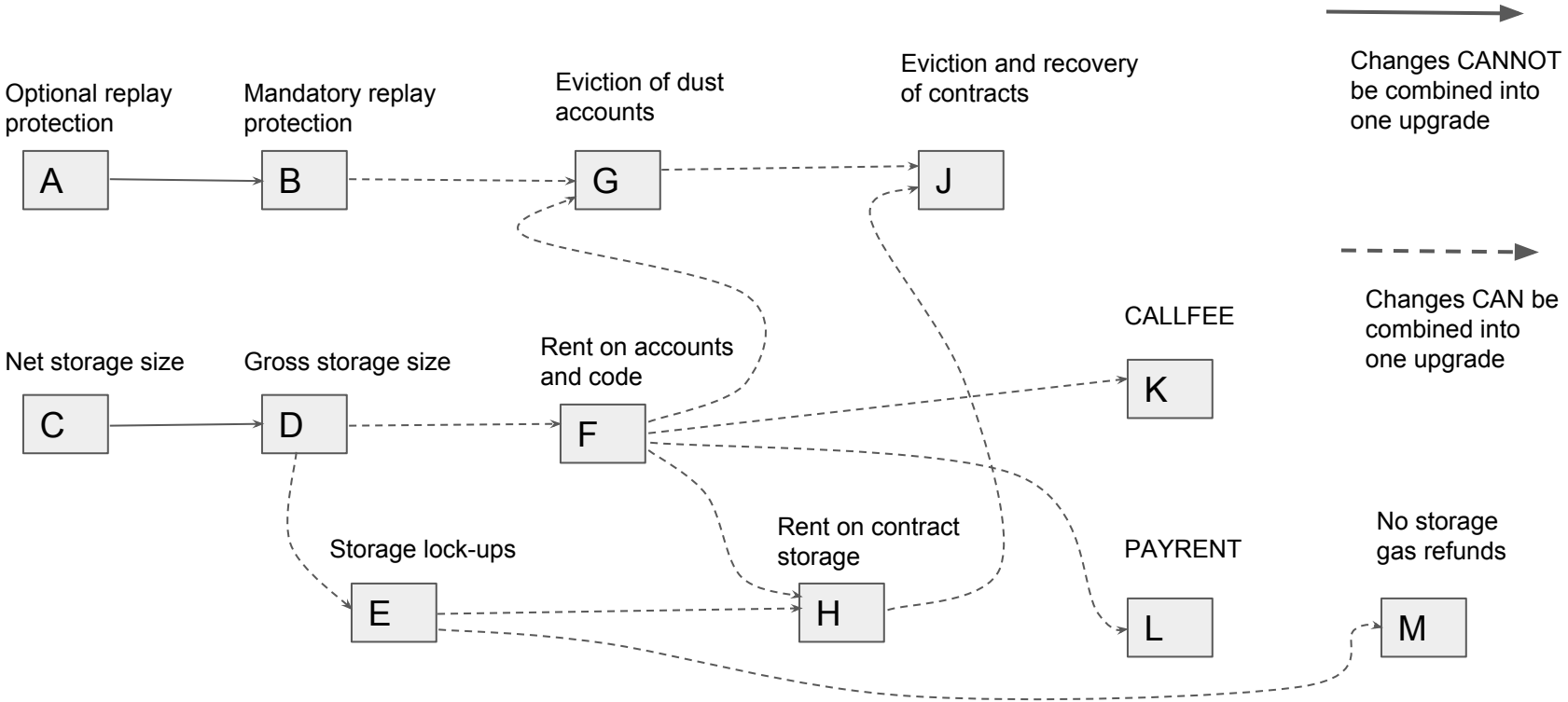
- Possibility to exempt important contracts from storage rent by burning ETH prior or after the introduction of rent
- Temporal replay protection is preferred over the variant with non-zero nonces
- Rent price is assumed to be fixed, floating rent can be introduced in later versions of the proposal

Organisation of this proposal

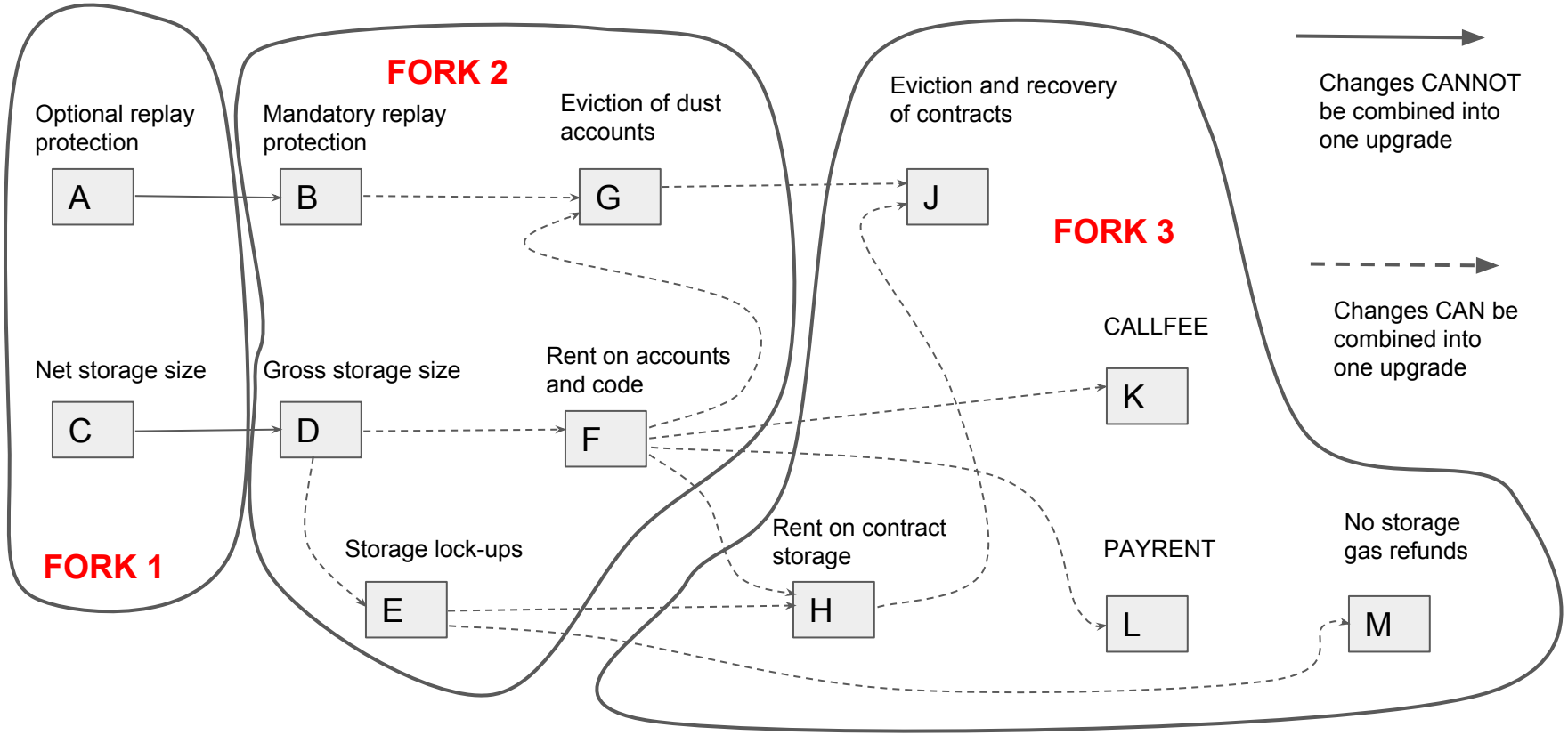
Proposal consists of the change cards, each given a capital letter, with the intention of each change potentially becoming an EIP (Ethereum Improvement Proposal). There are dependencies between the changes.

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

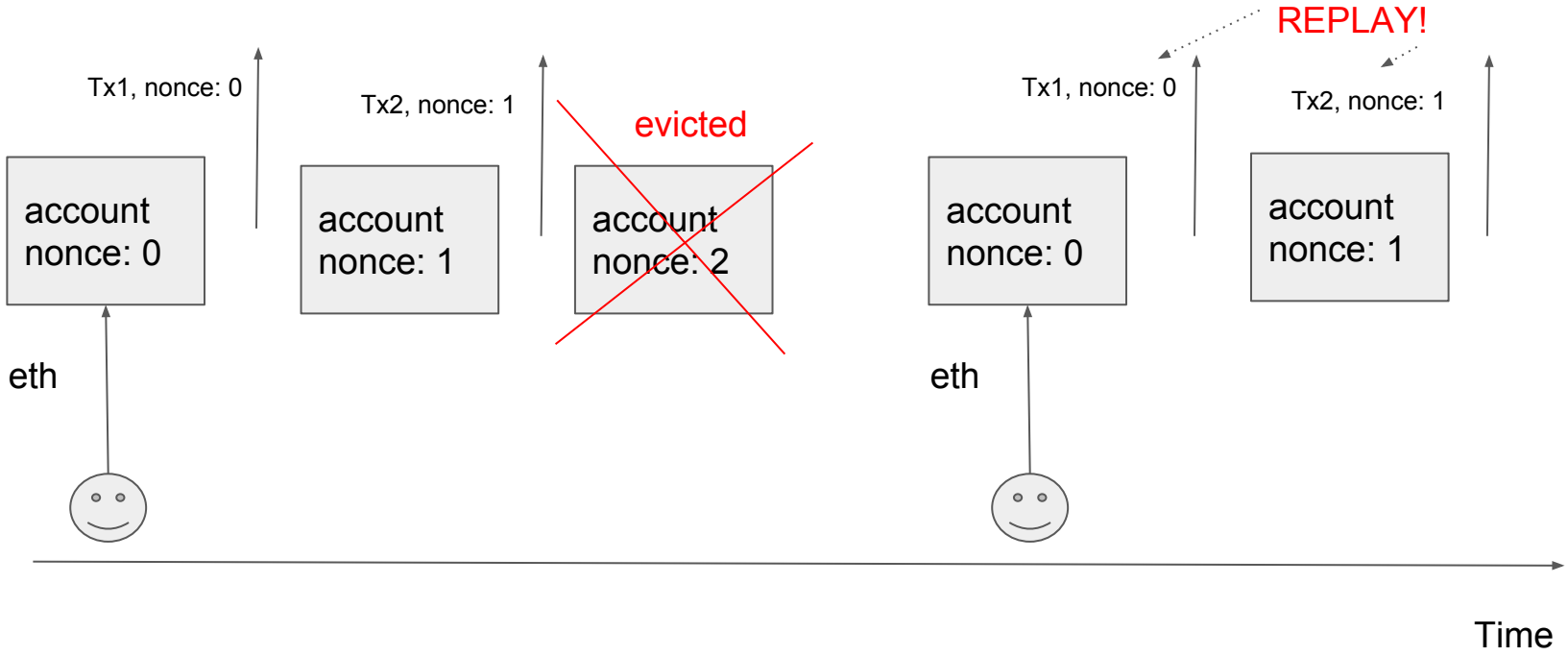


Potential timeline (in hardforks)



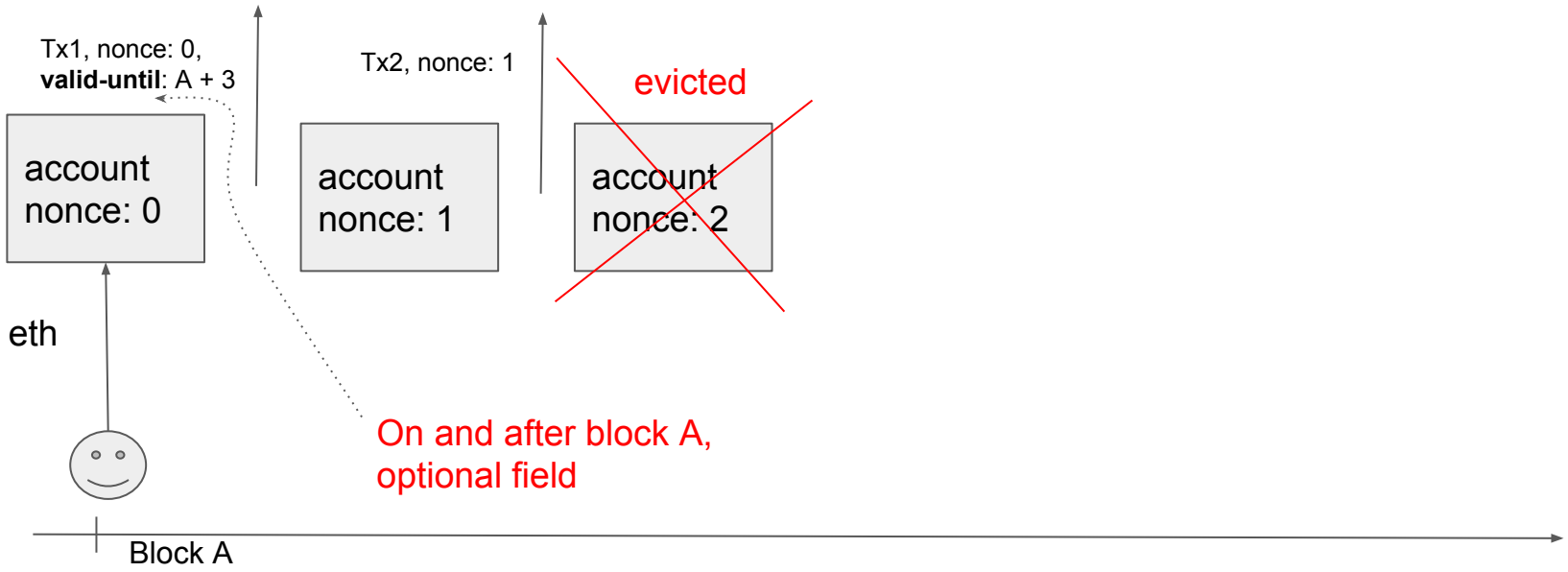
Need for replay protection for new accounts

Eviction creates a replay problem when account is recreated



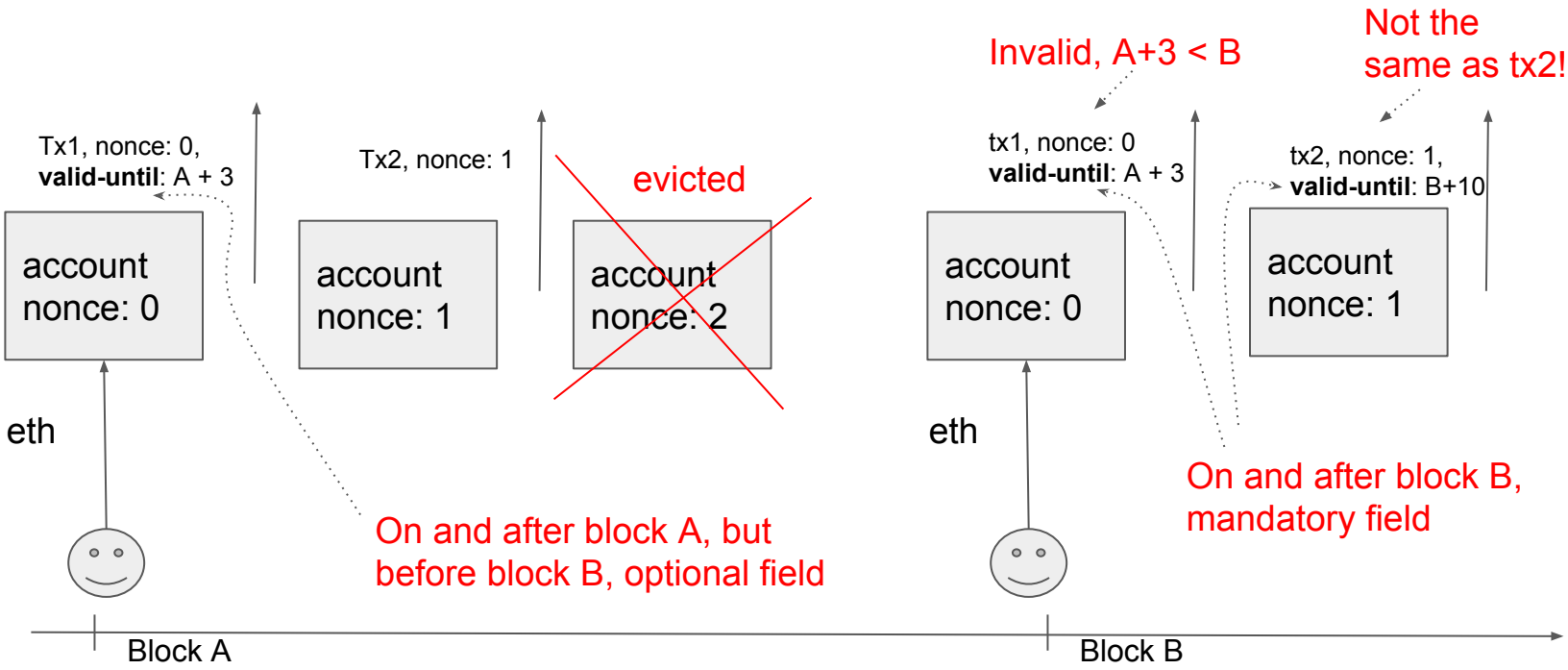
Change A, optional temporal protection

EIP draft: https://github.com/ethereum/EIPs/pull/1681



Change B, mandatory temporal protection

EIP draft: https://github.com/ethereum/EIPs/pull/1681



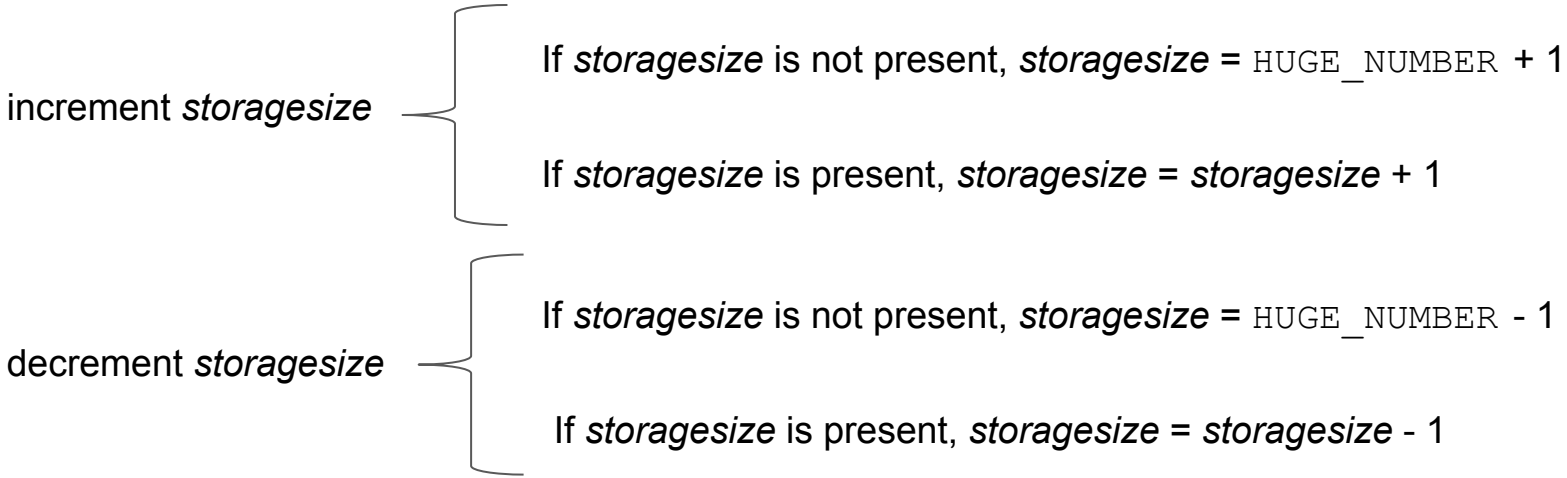
Change C, net contract size accounting

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 `HUGE_NUMBER` - <storage size at block C>

Change D - 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 it is value is more than `HUGE_NUMBER-<size_at_C>`, decrease it by `HUGE_NUMBER-<size_at_C>`

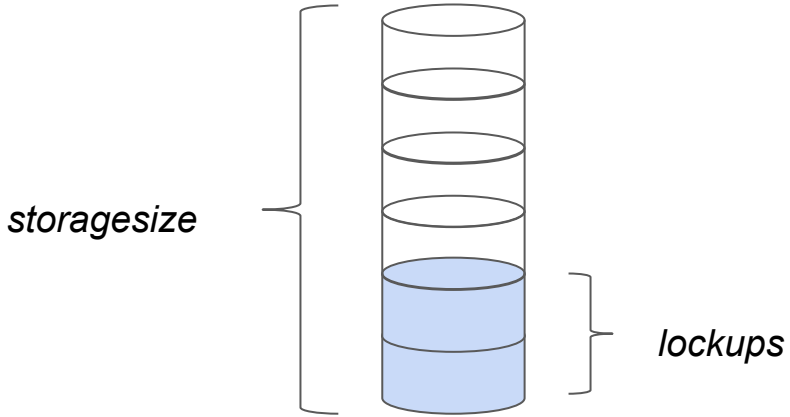
Change D - gross contract size accounting

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 it is value is more than `HUGE_NUMBER-<size_at_C>`, observed value is *storagesize*-`HUGE_NUMBER-<size_at_C>`

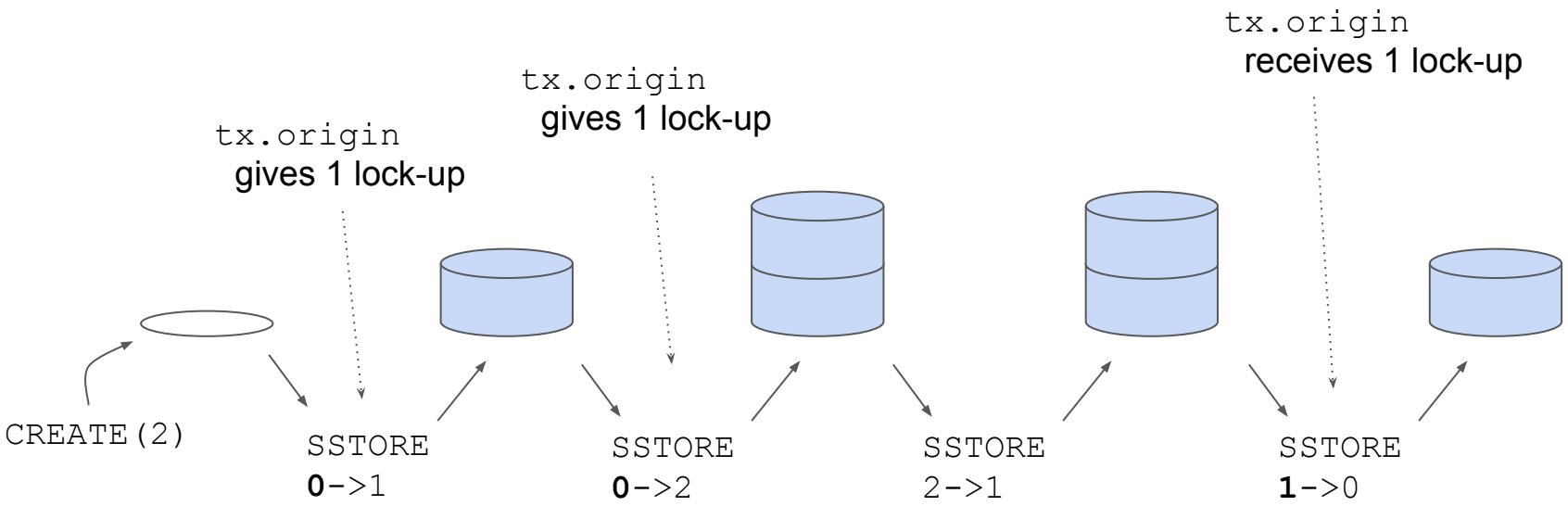
Change E - storage lock-ups

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 amount locked up



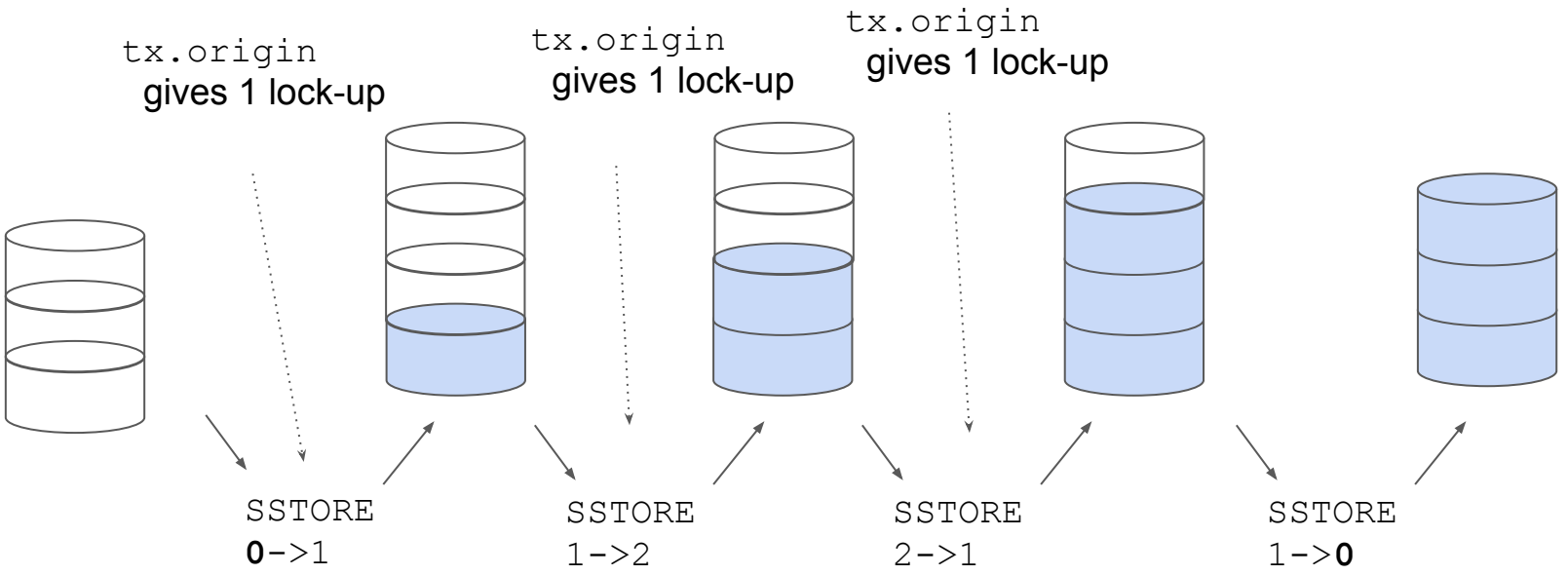
Change E - storage lock-ups

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



Change E - storage lock-ups

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



Change E - storage lock-ups

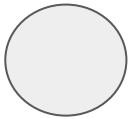
On and after block E, each contracts gets an additional field, *lockups*. Semantics of operation `SSTORE (index, value)` changes as shown below. Value of `[index]` at the beginning of transaction (not a single execution frame) is `original`. Value of `[index]` prior to the execution of `SSTORE` is `current`.

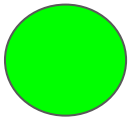
Without loss of generality, we will analyse only three possible values of `original`, `current`, and `value`: **0**, **1**, and **2**. “**0**” represents zero (value that can be removed from the state), “**1**” and “**2**” represent two distinct non-zero values (it does not matter what they are, all that matters is that they are no-zero and not equal to each other).

Change E - storage lock-ups

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

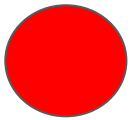
Ground state - no change in the state of the contract storage





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 E - storage lock-ups

It is quite straightforward to map all possible values of triple (`original`, `current`, `value`) 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 

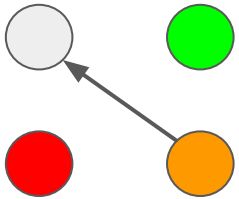
Change E - storage lock-ups

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

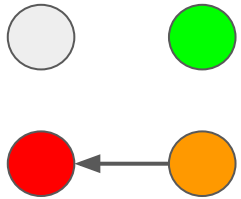
`original` -> `v-value`	**0**	**1**	**2**
0	ground 	removed 	removed 
1	new 	ground 	changed 
2	new 	changed 	ground 

Change E - storage lock-ups

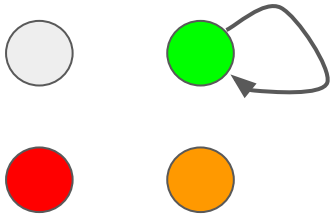
For example, the triple (`original`=**1**, `current`=**2**, `value`=**1**) represents transition:



Triple (`original`=**1**, `current`=**2**, `value`=**0**) represents transition:



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



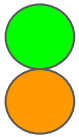
Change E - storage lock-ups

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:

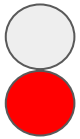
1. Contract pre-existed before introduction of lockups and contains some storage that does not have corresponding lockups, i.e. *lockups* < *storagesize*
2. Contract either created after introduction of lockups, or already “caught up” with the lock ups, i.e. *lockups* == *storagesize*

Change E - storage lock-ups

lockups < *storagesize*



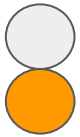
balance(`tx.origin`) -= lockup_price
lockups ++



no-op

lockups == *storagesize*

balance(`tx.origin`) -= lockup_price
lockups ++



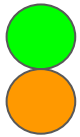
no-op

balance(`tx.origin`) += lockup_price
lockups --

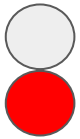
Change E - storage lock-ups

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

lockups < *storagesize*

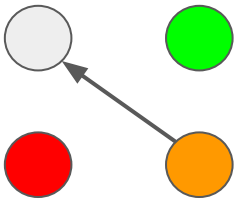


balance(`tx.origin`) -= lockup_price
lockups ++



no-op

(`original`**=1**, `current`**=2**, `value`**=1**)



Semantics:
balance(`tx.origin`) += lockup_price
lockups --

Change E - storage lock-ups

Lock-ups 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 lock-ups are. Therefore, it is easier to treat lock-ups 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.

Amount of wei can be deducted from `tx.origin` and locked up is only limited by the possible number of `SSTORE` operations within given gas limit. If this does not provide enough safely, transaction format might need to be changed to add an extra field *maxLockups*, which will put the cap of amount of wei that can be locked up during the transaction.

Change F - fixed rent on accounts

On and after block F, every account gets 2 extra field: *rentblock* (last time rent was calculated), with default = block F, and *rentbalance*, which can be negative.

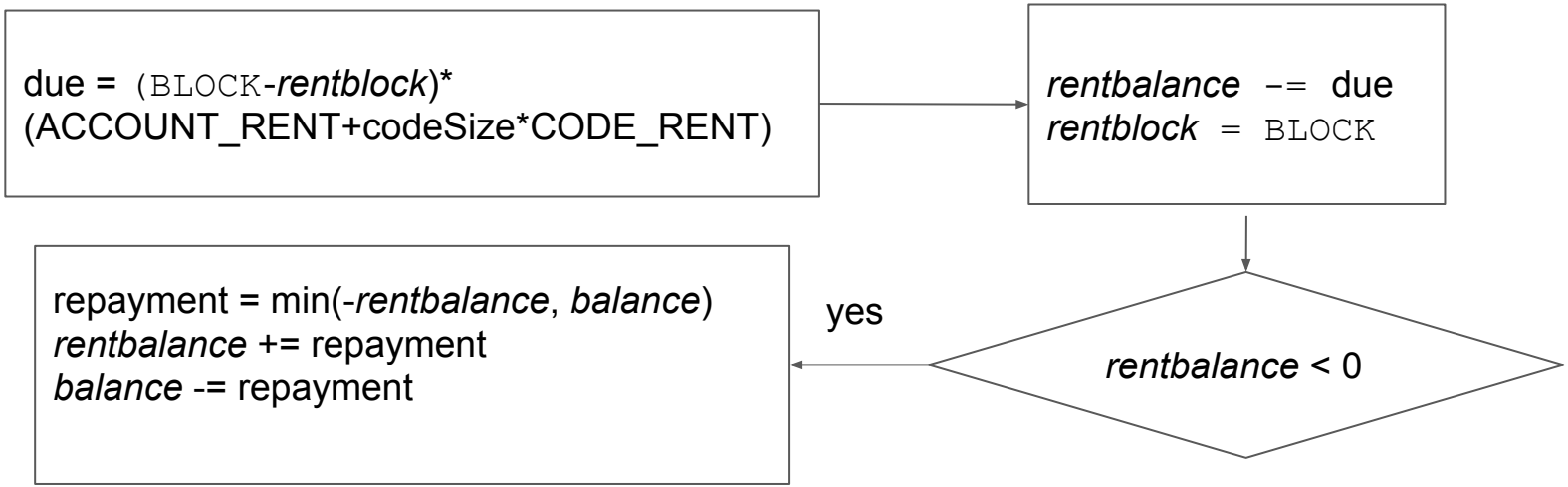
Two rent constants are defined:

1. ACCOUNT_RENT - rent charge in wei per account per block, for example, $2*10^9$.
2. CODE_RENT - rent charge in wei per byte of code per block, for example, $1*10^7$

Pre-compile contracts are exempt from rent.

Change F - fixed rent on accounts

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



Change F - fixed rent on accounts

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

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

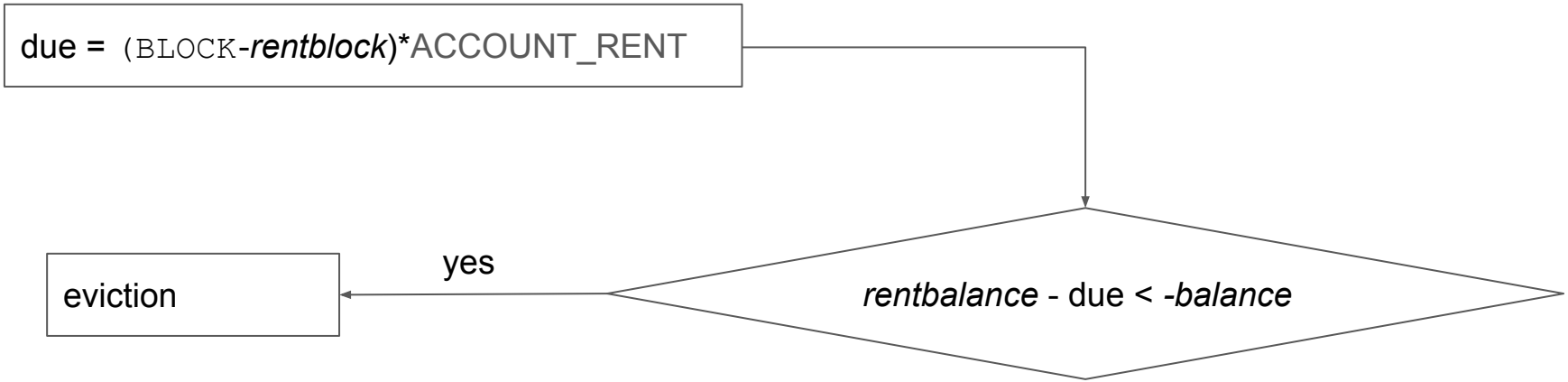
Rent calculation happens before stage (4), because it modifies state, but after stage (3), because stage (3) can increase the set of modified accounts.

Change G - Eviction of dust accounts

Dust accounts are defined as accounts with the codeHash == keccak256(nil), which is `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470` and with the balance of 0 wei.

Change G - Eviction of dust accounts

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



Change H - fixed rent on contract storage

If lock ups are present, then the rent for contract storage is only charged on the difference *storagesize - lockups*. That means that contracts created after the introduction of lock-ups, as well as contracts that “caught up” with the lock ups through usage, only pay rent on their account and code.

A new rent constant is introduced:

STORAGE_RENT - amount of wei charged for 1 item of storage without lock-ups, per block

Change H - fixed rent on contract storage

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

due = `(BLOCK-`*rentblock*`)` * (
STORAGE_RENT * (*storagesize* at the start of the block - *lockups* at the start of the block*)* +
ACCOUNT_RENT +
CODE_RENT * *codeSize* at the start of the block)

Values of *storagesize, lockups,* 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 D

Change J - Eviction and recovery of contracts

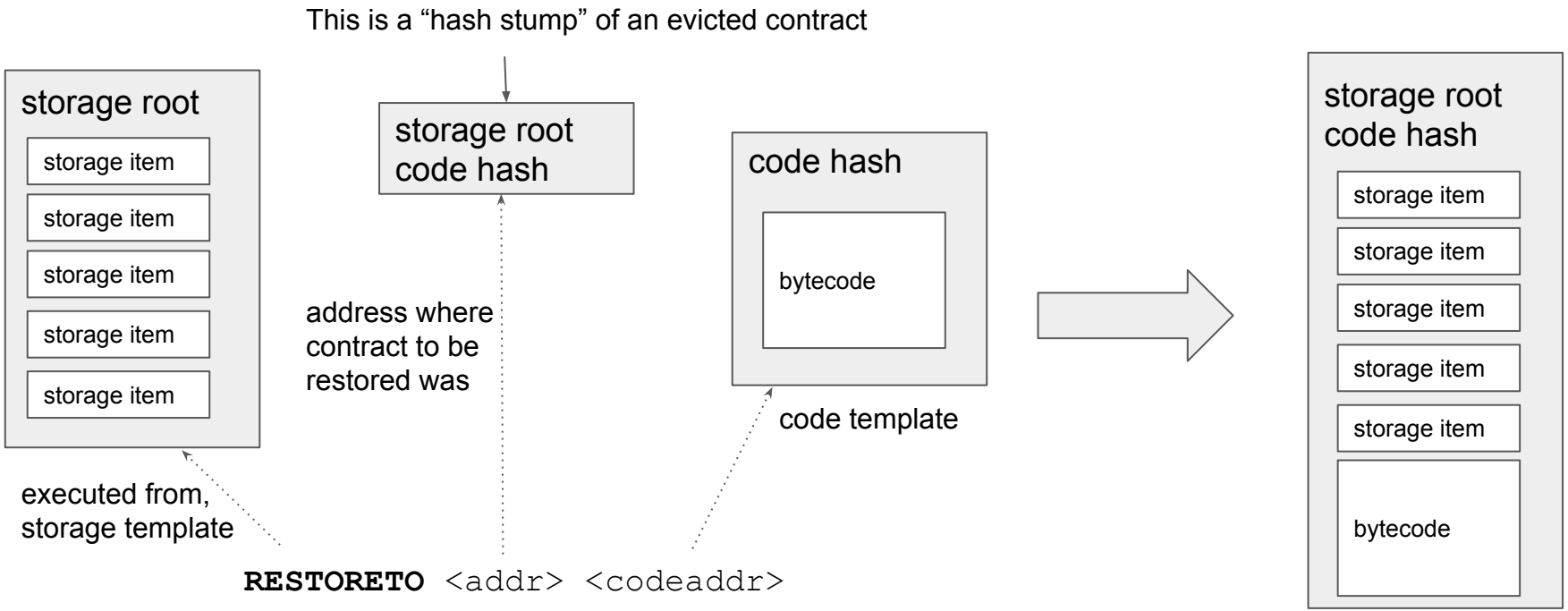
Eviction of contracts happens upon “touching” them in a transaction. Unlike the eviction of a non-contract account, eviction of a contract leaves a “hash stub” in the state. This hash stub can be later used to reconstruct the contract if it is still needed.

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 J - Eviction and recovery of contracts

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



Change K - 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 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`

There can be a way for waive the calling fee if the *rentbalance* is above some threshold, but this needs to be researched.

Change L - 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.

Change M - No storage gas refunds

Currently, `SSTORE` opcode gives gas refunds when storage is cleared. This refund is capped by the half of gas spent within the transaction.

Since release of locked up ETH represents much more straightforward and effective way of incentivising clearing the storage, storage gas refunds should be removed.