Suppose that you have an account or storage object (eg. a cryptokitty), and you want to make sure that the witness for it will be available for a very long time (eg. 20 years). In the fully stateless model, witnesses for very old data may disappear.

One way to solve this problem is via a specialized storage market: users pay third-party providers to store, and keep up to date, witnesses for some set of accounts or other state objects. This can be done with a credible commitment scheme: users send 0.XX ETH into a smart contract, from where they can never recover the ETH, but they can

at any point create a signed message to unlock some account A, at which point that ETH will be given to the sender of the next transaction which touches account A (the mechanism is simple: getbalance(A); send(msg.sender, XX)

; the idea is that actually processing a transaction that gets the balance of A requires the merkle branch for A to be in its witness). Sellers see that there are funds locked up, and so are willing to store the branch taking the gamble that eventually, payments for at least some of the branches they store will be available.

There are a few tradeoffs in this kind of mechanism

1: Can anyone claim a bounty, or only a few pre-specified storers?

In the first version, when the sender publishes a signed message that they want the witness for account A, anyone can claim the bounty. In the second version, we imagine that there exists a billboard, where anyone can assign themselves as a storer, and the transaction sender randomly selects perhaps 100 storers from the billboard. This seems more restrictive, but has the advantage that:

- It is easier to measure what percentage of storers actually provide that data after some amount of time
- Those 100 storers may be more willing to keep track of the witness, as they know that they are more likely to be the ones that will actually claim the reward.

Currently, I lean toward (1) for simplicity, and an intuition that the second argument may fall apart in an equilibrium model (if the fact that there are too many storers makes storing unprofitable, and this makes storers leave, then that makes storing profitable again).

2: How do we actually write the claiming contracts, and who stores and keeps up to date the witnesses for the claiming contracts?

In order to avoid creating an infinite descent problem, the answer would obviously have to be: the storers would have to store and keep up-to-date witnesses for both the accounts the users want to store and the claiming contracts.

To maximize efficiency, we can imagine each user having one such contract, and storing a Merkle tree mapping {account: (amount of wei to pay for uncovering the branch, claimable_from)}

, where claimable_from

is the starting block number at which the payment can be claimed. The user or the storers can store the mapping, and the storers would store and keep up-to-date the branch for the contract.

The code for the contract would be something like:

accounts: (payment: wei_value, claimable_from: num)[address]

@payable @public def add(addr: address): self.accounts[addr].payment += msg.value

@public def activate(addr: address, sig: bytes <= 400): assert self.is_sig_valid(sha3(addr), sig) assert self.accounts[addr].claimable_from == 0 # Wait 20 blocks, to prevent user from taking # back their own money too easily self.accounts[addr].claimable_from = block.number + 20

@public def claim(addr: address): assert self.accounts[addr].claimable_from > 0 assert self.accounts[addr].claimable_from <= block.number x = get_balance(addr) send(msg.sender, self.accounts[addr].payment) self.accounts[addr] = {0, 0}