- See Common classes of contracts and how they would handle ongoing storage maintenance fees ("rent")for the earlier equivalent of this post for per-storage-slot maintenance fees ("rent")

- See The Stateless Client Concept for a description of the "stateless client concept".

A conversion to contract statelessness would look as follows. Users would be required to specify as part of transaction data

a list of which storage keys of which contracts transactions can access, along with Merkle proofs for those storage keys; they would be charged some gas (eg. 10 gas per byte) for these Merkle branches. Attempts to access storage keys outside of the specified values would lead to an immediate VM exception. Miners and other full nodes would no longer need to save contract storage tries. Users would still be free to access all data other than storage keys without pre-specifying it. The scheme can also be modified to exclude, for example, storage keys 0…31 of every contract, allowing contracts with a limited amount of storage to be accessed arbitrarily.

If a miner receives a transaction that affects some storage trie and includes a Merkle branch, and it receives a block that modifies that storage trie, then we know that it is always possible for the miner to merge data from these two Merkle branches to construct and include a Merkle branch that proves the same storage value against the new state root.

General security of storage key access list

Before the Tangerine Whistle hard fork there was a maximum call stack depth of 1024. An attacker calling a contract could call themselves 2013 times first, making contract calls fail internally. Even today, gas limits may present a similar issue. Hence, for security, especially older contracts would have already needed to consider the possibility that child calls that they make will fail.

Tokens (ERC20)

Effect of statelessness

- none to users; however, other

contracts sending ERC20 tokens to dynamically determined addresses would be more difficult. For example, a contract that sends ERC20 tokens to the block coinbase would be impossible, as one cannot predict what the coinbase is ahead of time.

Solution

- if sends to dynamically determined addresses are desired, the contract could be designed to save partially completed sends at a user-specified index (checking that index has not already been used); the recipient could then send a transaction to claim the funds.

Cryptokitties

Effect of statelessness

- currently could lead to large gas consumption increase because addresses of new kitties are dynamic (see uint256 newKittenId = kitties.push(_kitty) - 1

in the contract), so transactions could need to specify a wide range of possible storage keys to get processed successfully.

Solution

- let users pick a new kitty ID, and use that ID unless it has already been claimed (in which case the transaction fails; but that's almost impossible if IDs are full 256-bit numbers)

Multisig wallets

Effect of statelessness

- currently could lead to gas consumption increase in theory because new operations get added to a list and so the index of the next list position is dynamic. Not an issue in practice because each multisig wallet has few users (unless someone spams the wallet as an attack; then a high-gasprice transaction specifying a large list of possible indices would be required)

Solution

- generate proposal ID based on hash of proposal data, store as map instead of list

Stateless multisig wallets

Effect of statelessness

- some classes of multisig wallets do not store state except for a list of owners and a sequence number (eg:this Vyper multisig). These are not vulnerable to attacks because they have O(1) storage, and a transaction can simply specify that it wants to access the entire storage tree.

ENS

Effect of statelessness

- very little; same theoretical concerns as ERC20s but likely to be even less of a deal in practice.

On-chain order books

Effect of statelessness

- depends on how they are constructed. Anything where orders are added to an in-protocol data structure (queue, heap…) is potentially problematic.

Solution I

- open orders become contracts.

Solution II

- orders are stored in a Merkle priority queue; accepting an order involves simply providing Merkle proofs of the pop

operation. Note that this also solves the on-chain sorting problem.

Smart contracts representing agreements

Effect of statelessness

- generally few. However, contracts involving large numbers of users in some cases will become more difficult to use. For example, consider this crowdfund contract that, in the case of a failed campaign, runs a loop to refund 30 users at a time. Multiple invocations of the refund transaction may interfere with each other as the indices of the users that get refunded would change.

Privacy-preserving contracts (mixers, anonymous voting, etc)

Effect of statelessness

- likely none, because users know ahead of time which I

value they are submitting, and a record can simply be placed at that position to show that that I

value will no longer be usable.

In general, effects on existing contracts seem to be much less serious than rent. With rent, contracts that have any O(n)-sized storage at all (mappings or

queues) are vulnerable to griefing attacks; with statelessness, in general terms mappings are unaffected but queues/lists are affected. However, even in the case of queues/lists, an application can still safely handle one user per block without disruption.