Computation on the Edges of Plasma-Blockchains

Christian Reitwießner
chris@ethereum.org

Abstract

We propose a programming paradigm to be used for smart contract blockchain system arranged in a tree-like structure as in the plasma [JP17] system. We show how smart contracts can be written so that invariants global to the whole system are kept, even if users only monitor certain sub-chains. This should provide massive scalability with only small drawbacks in security.

1 Introduction

The plasma system [JP17] defines a structure of interconnected blockchains arranged in a tree structure that promises scalable smart contracts. One of the key ideas there is that each of the blockchains regularly store their current block hash in their parent chain so that users can challenge potentially invalid child state transitions in the parent chain. This model is secure not because of a difficult proof of work (the chains would use proof of stake or even a fixed validator set), but rather because users watch chains they have a stake in and thus will challenge invalid state transitions, potentially escalating as far up as the trusted main chain.

Here, the scalabality does not come from the fact that blockchains are relieved from their load by creating a big number of smaller chains and moving the transactions there. Scalability is only achieved once a user does not have to verify every single transaction that is sent to the system. If, for example, a user only cares about a single smart contract that resides in a single chain that is a leaf of the system, it is sufficient for the user to verify this leaf and all nodes on the path to the root chain. If a transaction is commited by means of block hashes all the way up to the root chain and there is no invalid state transition in the chains on the way up to the root, the user can be reasonably sure that the transaction cannot be declared invalid by other users, essentially halting at the last "known good" state.

This system still does not solve the scalability problem: As long as the smart contract only "lives" inside a single blockchain, it can merely process a limited amount of transactions. While this might be enough for some use-cases, a token contract can easily reach this limit.

The system would scale, if the token contract exists on all of the blockchains and it is possible to move tokens up and down the tree. Users would have accounts in only one or perhaps some of the chains and watch the paths to the root from chose chains. In such a simple model, an attacker could just select a chain that is mostly unused, take it over, create an invalid state transition that creates tokens out of thin air and then move these tokens up to the root. If nobody is watching the attacked chain (or the attacker can turn off their computers or censor their transactions), the attacker is safe as soon as he or she is able to move the tokens far enough up.

Creating tokens out of thin air is a violation of the assumed invariants of a token contract. Unfortunately, in contrast to the other invariant "nobody but me can spend my tokens", this is a global invariant. Luckily, this global invariant can be split in a way, such that only enforcing it locally will still enforce it globally, or at least in the root chain:

We track the sum of the token balances of each direct child chain in the smart contract of the parent chain and enforce that changes to these numbers always have to come with a respective amount of tokens being moved from the child to the parent or vice-versa. In this situation, an attacker can still create tokens in unwatched child chains, but he or she can only move as many tokens out of these flawed chains as is their total balance. For users that are not interested in these chains, the situation would not change: For them, someone took out tokens from a pool, but it is not relevant who did it. In effect, the attacker of course steals tokens from users who that have accounts in the attacked child chains, but at least the impact of the attack is confined to chains that are not properly watched. In turn, this means that you have to constantly watch all chains you have tokens in. But there is also a solution to this: If you do not want to move your tokens very often, you can transfer them to a chain further towards the root. These chains likely have higher transaction fees (which is not relevant if you just want to park your tokens), but also provide higher security because they are watched by more people.

In the next sections we will give examples of smart contract systems, how they can be distributed among a plasma system and will also go deeper into how the synchronisation between parent and child chains is implemented.

2 Assumptions

We start by defining a simplified view of the plasma system.

We abstract away the fraud proof mechanism and blockchains in general.

There is a number of *chains*, each of which is modeled as a computing node. Some chains are malicious, meaning they can have invalid state transitions, ignore transactions or withhold information. The chains are arranged in a tree and have a communication channel to thair parent and their children, if they have some. We assume that the root chain is not malicious.

There is a certain number of users. Some users are malicous. We assume that each user can *watch* a limited number of chains. Non-malicious users watch some chains and all chains on the path to the root from these chains.

We also assume that if a user watches a chain, it cannot have any invalid state transitions for that period of time, unless the user is also malicious or the parent chain can have invalid state transitions.

State transitions in individual chains are unrelated and happens asynchronous. Communications between chains are synchronisation points.

3 Token Contract

TODO: I wanted to give a formal abstract treatment first, but it is probably better to argue alongside the contract.

The following smart contract is replicated on all chains in the system. The smart contract language used follows the syntax and semantic of Solidity [sol17], but it has one additional feature: Functions can be marked “edge”. Such functions are executed as part of a transaction sent to an “edge” of the tree instead of a sigle chain. Parts of the code of these functions are executed on the relative parent and other parts on the relative child in sequence. The actual compiled smart contract will use logs and Merkle proofs for synchronisation. Inside an “edge” functions, the identifier “child” is an integer (0 or 1 for two children) identifying the child relative to the parent.

In the event where a child chain is declared faulty, the child parts can also be executed in the parent chain or recursively in any chain on the way up to the root (updating the root hash of the child chain stored in the parent chain and requiring relevant Merkle proofs for execution). A child that does not react to edge transactions after a certain time is declared faulty.

```
contract Token {
  mapping(address => uint) balance;
  uint[2] childBalance;

  // Regular transfer between two accounts in the same chain.
  function transfer(address recipient, uint amount) {
    require(balance[msg.sender] >= amount);
    balance[msg.sender] -= amount;
    balance[recipient] += amount;
  }

  // Moves tokens from a certain account in the parent to the
  // same account in the child chain.
  edge function transferToChild(amount) {
      parent {
        require(balance[msg.sender] >= amount);
        balance[msg.sender] -= amount;
        childBalance[child] += amount;
      }
      // After successful completion of the parent part,
      // the child part can be executed in the child.
      child {
        balance[msg.sender] += amount;
      }
```

```
  }

  edge function transferToParent(amount) {
    child {
      require(balance[msg.sender] >= amount);
      balance[msg.sender] -= amount;
    }
    parent {
      require(childBalance[child] >= amount);
      childBalance[child] -= amount;
      balance[msg.sender] += amount;
    }
  }
}
```

Property 3.1. *If a non-malicious user watches all chains where he or she owns tokens and only accepts to receive tokens on chains where he or she has either validated the full history of the chain (including the path to the root) or checked that sum of all balances (including child balances) matches the respective amount in the parent chain (including the path to the root), then the user can always move these tokens to the root chain.*

Proof. First, let us observe that if such a user owns accepted tokens in a chain, the user watches this chain and its parents and thus the code will executed exactly as stated.

Furthermore, note that in each situation where a balance of a user (as opposed to a child chain) is decreased, the address "msg.sender" is used and thus this only happens if the user intends to do so.

If a child is ignoring edge transactions, it can be declared faulty. Since the user has been watching the chain, the user can provide Merkle proofs to move his or her tokens to the parent by executing the edge transaction only in the parent, potentially recursively until a non-faulty chain is reached.

If the child does not ignore edge transactions and is not declared faulty, all contracts on the path to the root execute as written. Then, the only reason the tokens cannot be moved to the root is that some call to "transferToParent" fails. Since the same value of "amount" is subtracted in the child chain and added in the parent chain, the reason for such a failure has to be a failed "require" call.

A) The call fails in the child part. The call can only fail in the child if the user wanted to send more tokens that his or her balance is, but the user would send exactly "balance[user]", so this cannot happen.

B) The call fails in the parent part. In this situation, we have to have "childBalance[child] < amount". Remember that the user either verified the full history of the chain or checked that the balances in the child sum up to "childBalance[child]" in the parent. At least at synchronisation points, this property is an invariant also for all other functions. Since "child.balance[msg.sender]

== amount" at the beginning of the function call, we have that "childBalance[child] $\geq$ child.balance[msg.sender]" and thus a contradiction.

Since both situations are impossible, the user can move the tokens to the root chain. $\square$

4 Second Example

5 Details of Edge Synchronisation

Transactions always contain a unique ID of the chain they are sent to to avoid replays in other chains.

References

[JP17] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts. `https://plasma.io/plasma.pdf`, 2017.

[sol17] Solidity. `https://solidity.readthedocs.io/`, 2017.