small 1-to-N blockchain transactions

PePER*

Abstract

The scaling problem is the challenge of processing more transactions per second in a blockchain while still maintaining the decentralized qualities of the technology. In unscaled blockchains a bottleneck is created during times of high transaction load, which causes transactions to be expensive and slowly processed. This paper tackles those two by-products problems.

We focus on creating small 1-to-N transactions, which are cheaper and faster to make, and suggest two types of such transactions. The size is independent of the complexity of the transactions; it is the same for both simple payment transactions or complicated smart contracts.

The first type of transactions we describe is called "percentage tree transactions", and requires $O(Log(N))$ data to be sent to each of the N recipients. The second type, called "RPS (Reusable Payment Scheme) transactions", requires $O(N)$ data to be sent to each of the N recipients, but has the advantage that multi-payments can be made using the same scheme.

We provide implementation written in Solidity for Ethereum, and discuss possible applications.

1 Introduction

A standard 1-to-N transaction in a blockchain is big in size if N itself is big. This results in transactions that are expensive to make (for the sender) and slow to be processed by the blockchain, as they need to be divided into a few blocks for N big enough.

This paper suggests an implementation of small 1-to-N transactions. The size is independent of the complexity of the transactions; it is the same for both multisig transactions or simple payment ones.

We provide two types of small 1-to-N transactions. The first, called "percentage tree transactions", require $O(Log(N))$ data to be sent to each of the N recipients. The second, called "RPS (Reusable Payment Scheme) transactions", require $O(N)$ data to be sent to each of the N recipients, but has the advantage that multi-payments can be made using the same scheme.

The tradeoff of both types of transactions is that while they are small to make, subsequent transactions have bigger space and computation complexity than regular ones.

Both types of the small 1-to-N transactions are based on Merkle trees. The first type uses a version of Merkle trees we call "percentage trees", while the second uses regular Merkle trees. A 1-to-N transaction writes, among other things, the root of a tree to the blockchain, with an unlocking script that is built around the tree structure.

We give an implementation in Ethereum for percentage tree transactions, and discuss issues of possible implementations in Bitcoin.

Two examples of possible applications are given: a fast creation of huge Lightning network payment hubs and a continuous-style salary payments of a big company to its workers.

2 History

Blockchain technology was invented in 2008 by Satoshi Nakamoto in the Bitcoin whitepaper [9]. The main application of this technology is in cryptocurrencies, with 1988 cryptocurrencies documented in CoinMarketCap in September 2018 [3].

*Paul Peregud (OmiseGo) and Eyal Ron (Cryptom Technologies UG)

As cryptocurrencies gained popularity and the amount of transaction increased, the scaling problem of processing more transactions per second while still maintaining the decentralized qualities of a blockchain became an important one. Both protocol solutions for the scaling problem [2, 1, 11] and offchain solutions [13, 12] were suggested.

In unscaled blockchains a bottleneck is created during times of high transaction load, which causes transactions to be expensive (an average of 50 dollars per transaction in Bitcoin in December 2017 [10]) and slowly processed (see how Ethereum got clogged by Cryptokitties [6]). The algorithms in this paper are not solutions to the scaling problem, but rather to the two by-product problems of expensive and slow transactions. See the discussion of related work in Section 6.

3 Blockchain model

To be precise yet independent of a specific blockchain implementation, this section describes a simplistic model for blockchain transactions. While this model is incomplete and not meant to be implemented in real applications, it allows us to state results in a clear, accurate fashion.

Our blockchain transaction model contains two objects, coins and scripts. Transactions are messages that are written to the blockchain, and change the state of objects.

Each coin is associated to a script (also said to be "locked in a script"); this association is changed only by transactions. The action of changing the association of a coin is called "unlocking", and we say that the transaction that associated the coin to the script is "unlocked". Similarly, if someone has the information needed to transfer the coin, we say that they can "unlock" the coin (or the transaction).

If Alice broadcasted a transaction that changed the association of the coin from a script that Alice could have unlocked, to a script that Bob can unlock, we say either that "Alice transferred coins to Bob" or that "the transaction transferred coins from Alice to Bob".

To discuss, in Section 4.4 and 5.4, the relation between the transactions defined in this paper and regular transactions, we define the notion of equivalent transactions. Two transactions are equivalent for Bob, if the data that Bob possesses allows him to unlock the same amount of coins regardless which of the two transactions was added to the blockchain.

Scripts are libraries written on the blockchain that contain state variables (i.e. local variables that the script stores on the blockchain and determine the state of the script after each transaction), a constructor function and a collection of boolean functions. Each script has a unique ID (though we do not discuss how it is given). Scripts can access some data in the transaction that called the script, namely the following variables are available to a script.

- Transaction.amount: the amount of coins in the transaction,
- Transaction.toScript: the script to which the coins are transferred,
- sigs: the collection of signatures for the transaction.

We also define *stateless complete scripts*. These are scripts that have no state variables, and that a valid transaction from the script must transfer all the coins associated to the script. The 1-to-N transactions in this paper are limited to transaction that transfer coins to stateless complete scripts (see explanation in Remark 3).

A transaction in the blockchain does one or more of the following three things:

1. writes a script to the blockchain,
2. changes the value of state variables of a script,
3. changes association of coins from one script to another.

The kind of transactions we consider here are of the form

$$Transfer\big(fromScript.function, amount, toScript, sigs\big), \tag{1}$$

where

- $fromScript.function$ is some function in the script to which the coins are currently associated,

- $amount \geq 0$ is the amount of coin to transfer,
- *toScript* is the new script to which the amount will be associated. It can be either an ID of an existing script, or a code of a new script,
- $[sigs]$ is a list of signatures, signing all the parameters for the $Transfer$ function (besides the 'sigs' parameter itself, of course).

Both $fromScript.function$ and *toScript* may require additional parameters.

A transaction is valid if

1. $fromScript.function$ returns "True",
2. $amount \leq$ amount of coins associated to $fromScript$,
3. *toScript* is a valid script.

4 1-to-N transaction type 1: percentage tree transaction

This type, as its name implies, is a 1-to-N transactions based on percentage trees (see Definition 2).

The core idea is to make one transaction with some amount of coins, and then each subsequent transaction allows each of the N recipients to unlock their portion of this amount. To know how to divide the coins, we build a percentage tree such that each leaf of the tree shows the percentage of the amount belonging to the corresponding recipient. A percentage tree is built in a way where it is enough for the recipient to hold only the Merkle proof (see Definition 2) for its leaf, to ensure that he can unlock the correct amount of coins.

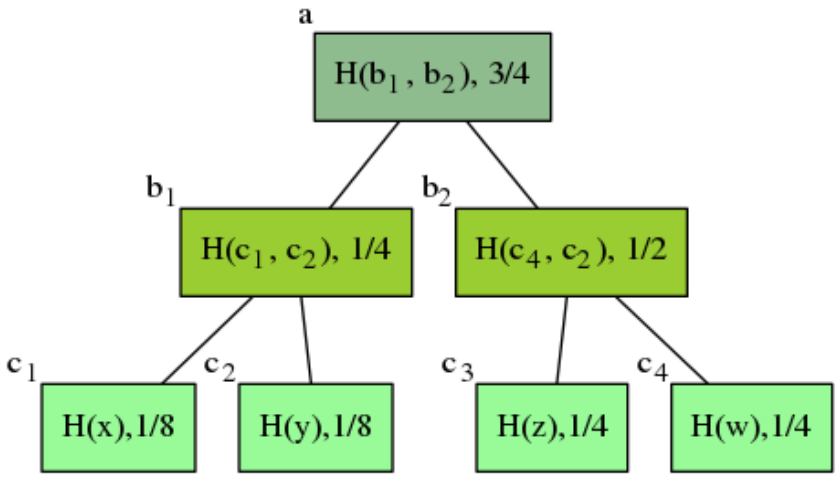
The letter H, in this section and throughout the rest of the paper, denotes a 'hash function' (we do not discuss which specific hash function is being used).

4.1 Percentage trees

Definition 1. A percentage trees is a Merkle tree where

- each node is a tuple of the form
$$\{hash, percentage\},$$
where the percentage is a rational number in $[0, 1]$,
- the sum of percentages of all leaves is at most 1,
- the percentage of each non-leaf node is a sum of the percentage of its children.

The figure below is an example of a percentage tree, with root $a = \{H(b_1, b_2), 3/4\}$.



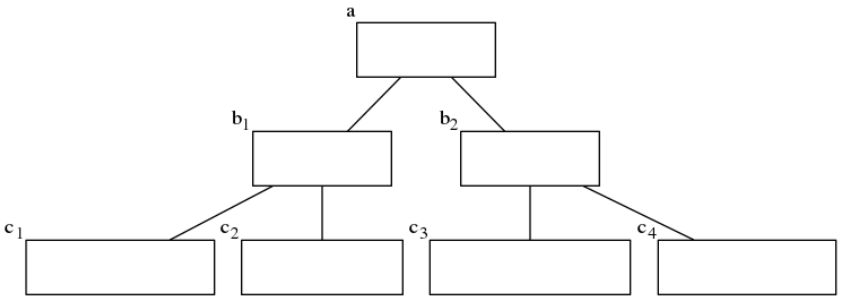
Definition 2. A Merkle proof for a leaf in a Merkle tree is the minimum list of all the nodes in the tree, such that a tuple of (leaf, Merkle proof, root) is sufficient to prove that the leaf belongs to the tree having this root.

4.2 Making a percentage tree transaction

Assume Alice wants to transfer $C_1 \ldots, C_N$ coins to stateless complete scripts $S_1, \ldots, S_N$, correspondingly. The two main components of a percentage tree transaction for those transactions are:

1. A percentage tree, where the hashes in its leaves are hashes of $S_1, \ldots, S_n$.
2. An N-bit: specifies which of the N transaction were already unlocked.

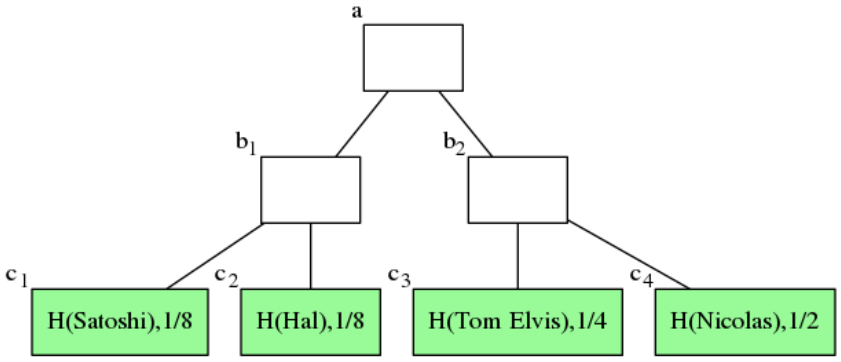
We demonstrate the construction of the percentage tree for such a transaction with an example. Begin with an empty tree.



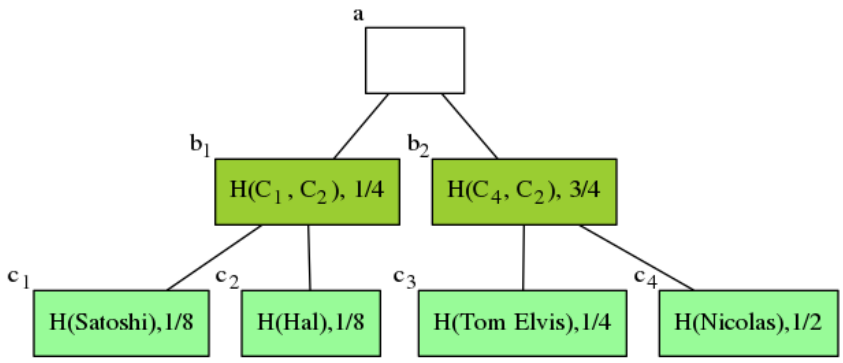
We create a tree that corresponds to the transactions that Alice wants to create.

- 1 coin to Satoshi,
- 2 coin to Hal,
- 4 coin to Tom Elvis,
- 16 coin to Nicolas,

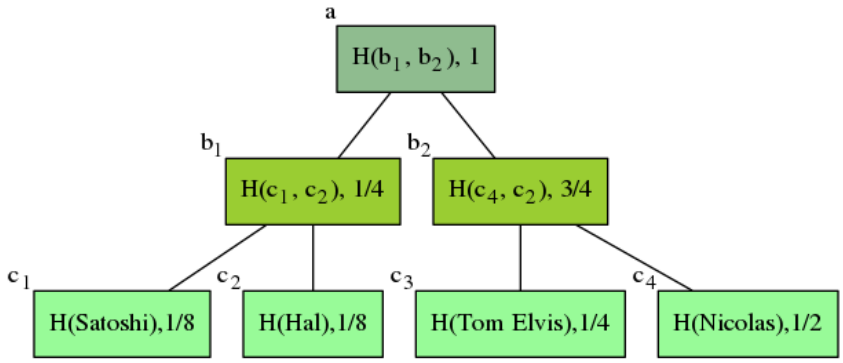
First fill in the leaves.



In the second layer, hash the leaves, and sum up their percentages.



To finish, calculate the Merkle root, note that the sum of percentage is not bigger than 1.



The protocol of Alice creating a 1-to-N transactions, each transactions transferring $C_1, \ldots, C_N$ coins to $B_1, \ldots, B_N$ (B signify "Bob") by locking it in stateless complete scripts $S_1, \ldots, S_N$ that the B_is can unlock, is as follows. First denote

$$A = \sum_{i=1}^{N} C_i.$$

Alice then creates a percentage tree, where the i-th leaf, L_i, has the form:

$$L_i = \{H(T_i), C_i/A\}$$

Alice gives each B_i the Merkle proof of L_i in the tree.

To make the percentage transaction, Alice broadcasts

$$Transfer\big(fromScript, A, scriptPercentageTx(r), AliceSigs\big),$$

where $fromScript$ is a script in which Alice has A coins locked, r is the root of the percentage tree, and the structure of the $scriptPercentageTx$ is described in Script 1.

Script 1 scriptPercentageTx(root)

```
1: state variable:
    2: Root := root //initial value
    3: WithdrawBit := bit vector of size N where all entries are 1
    4: Amount := Transaction.amount
5: function unlockTx(MerkleProof, unlockScript, percentage, [unlockScriptParams])
```

The function $unlockTx$ is described in the next subsection, in Function 3.

The precise protocol for making percentage tree transaction is given in Protocol 2.

Protocol 2 making a percentage tree transaction

1: **Scenario**: Alice makes N transactions, each transactions transfers $C_1, \ldots, C_N$ coins to to $B_1, \ldots, B_N$ by locking it in a stateless complete script $S_1, \ldots, S_N$ that the B_is can unlock
2: Set $A := \sum_{i=1}^{N} C_i$
3: Define leaves $L_1, \ldots, L_N$, each L_i is a tuple $\{H(S_i), C_i/A\}$
4: Build a percentage tree where $L_1, \ldots, L_N$ are its leaves, denote by r the root of the percentage tree
5: Send each B_i the Merkle proof for leaf L_i
6: Broadcast:
7: $Transfer\big(fromScript, A, scriptPercentageTx(r), AliceSigs\big)$,
8: where $fromScript$ is a script in which Alice has A coins locked

4.3 Unlock a percentage tree transaction

If Alice made a percentage tree transaction of amount A, then B_i received from Alice the Merkle proof, P_i, corresponding to leaf of L_i (that corresponds to stateless complete scripts S_i where C_i coins are locked). To transfer C_i coins to another script $toScript$, B_i broadcasts the transaction.

$$Transfer\big(scriptPercentageTx.unlockTx(P_i, S_i, C_i/A, [T_iParams]), C_i, toScript, [B_iSigs]\big).$$

The unlockTx function is:

Script 3 function unlockTx(MerkleProof, unlockScript, percentage, [unlockScriptParams])

```
1: if MerkleProof ∈ percentage tree with root == 'Root' then
2:   if {H(unlockScript), percentage} == MerkleProof.leaf then
3:     if unlockScript([unlockScriptParams]) == TRUE) then
4:       if i-th bit of withdrawBit == 1 then
5:         if percentage*Amount == Transaction.amount then
6:           0 → i-th bit of withdrawBit
7:           return TRUE
8:         end if
9:       end if
10:    end if
11:  end if
12: end if
```

Technical remark: recall that in line 3 in the in last script, the "unlockScript" also has implicit access to B_i's signature that were passed in the transaction.

The precise protocol is given in Protocol 4.

Protocol 4 Unlock a transaction

1: **input**: A percentage tree transaction with root R and amount A exists in the blockchain, where B_i has a Merkle proof M_i for a leaf that contains stateless complete scripts S_i with percentage C_i/A
2: Params := parameters for unlocking S_i
3: B_i broadcasts:
4: Transfer$\big($scriptPercentageTx.unlockTx$(M_i, S_i, C_i/A, [Params]), C_i$, toScript, $[B_iSigs]\big)$

Remark 3. *At this point it should be clear why the* 1*-to-N transactions are limited to stateless complete scripts. An unlock transaction from script S_i changes the i-th bit of withdrawBit to* 0*, meaning that no further transactions can be made from script S_i. If we didn't force the script to allow only transactions that transfer all the coins locked in it ("complete transactions"), then the coins which weren't transferred would be locked forever. Since only one unlock transaction is possible from script S_i, there is no need for S_i to have state variables.*

4.4 Analysis

Assume that $B_1, \ldots, B_N$ have the information to unlock scripts $S_1, \ldots, S_N$, correspondingly. Alice wants to transfer $C_1 \ldots, C_N$ coins to scripts $S_1, \ldots, S_N$, correspondingly. We analyze the following scenario.

Scenario 1: Percentage tree.

1. Alice makes a 1-to-N percentage tree transaction, where each leaf L_i of a the tree contains a script S_i and an appropriate percentage to represent the amount C_i.
2. Each $B_1, \ldots, B_N$ makes a transaction, and each transaction associates the $C_1, \ldots, C_N$ coins to new scripts $S_1^n, \ldots, S_N^n$ correspondingly.

Specifically, we compare the percentage tree scenario to the following scenario.

Scenario 2: Classical.

1. Alice makes N transactions, each transaction associates $C_1, \ldots, C_N$ coins to $S_1, \ldots, S_N$ scripts, correspondingly.
2. Each $B_1, \ldots, B_N$ makes a transaction, each transaction associates the $C_1, \ldots, C_N$ coins to new scripts $S_1^n, \ldots, S_N^n$ correspondingly.

Note that both scenarios differ only in the first step.

Recall from (1) that a transaction includes two scripts, fromScript and toScript. For step 1 in the scenarios we concentrate on calculating the complexity of toScript, while for step 2 we calculate the complexity of fromScript.

Theorem 4. *Let $Z_1^1, \ldots, Z_N^1$ be the sizes of scripts $S_1, \ldots, S_N$ correspondingly, and $Z_1^2, \ldots, Z_N^2$ be the size of data that B_i needs to unlock his coins from $S_1, \ldots, S_N$. Let $U_1, \ldots, U_N$ be the number of steps needed for B_i to unlock his coins from $S_1, \ldots, S_N$, correspondingly. The following claims are true.*

1. *The transaction in step 1 of Scenario 1 is equivalent to the N transactions in step 1 of Scenario 2.*
2. *The size of toScript in step 1 of Scenario 1 is $N + O(1)$, where $O(1)$ is the size of a hash of the hash function. The corresponding size in step 1 of Scenario 2 is $\sum_{i=1}^{N} Z_i^1$.*
3. *The size of fromScript in step 2 of Scenario 1 is $N * \log N + \sum_{i=1}^{N} \left(Z_i^1 + Z_i^2\right)$. The corresponding size in Scenario 2 is $\sum_{i=1}^{N} Z_i^2$.*
4. *The computational complexity of toScript in step 1 in both scenarios is $O(1)$. The computational complexity of the fromScript in step 2 of Scenario 1 is $\sum_{i=1}^{N} U_i + N \log N$, while the corresponding complexity in Scenario 2 is $\sum_{i=1}^{N} U_i$.*

Proof. To prove the first claim we note that in both scenario each B_i holds the data to unlock script S_i. Since after the classical scenario there are C_i coins associated to S_i, B_i can unlock C_i coins. In the percentage tree scenario B_i also has from Alice the Merkle proof for leaf L_i. So B_i can unlock also in this scenario C_i coins from the percentage tree transaction.

The rest of the claims are straightforward calculations. □

The theorem above shows that percentage tree transactions are smaller to make in size than classical N transactions, assuming that the size of a classical transaction is bigger than 1 bit. It also shows the tradeoff, both in size and computational complexity, of unlocking transactions.

5 1-to-N transaction type 2: RPS (reusable payment schemes)

Definition 5. *A reusable payment scheme (RPS) is a Merkle tree where each leaf is of the form*

$$\{H(S), P\},$$

where S is a stateless complete script and $P \in (0, 1]$ is a rational number (percentage), and the sum of P values in all leaves is smaller or equal to 1.

The second type of 1-to-N transaction is used for multiple payments. Reusable payment schemes are done in two steps. First, Alice creates a Merkle tree for an RPS and writes it on the blockchain. Second, Alice (or someone else) makes a 1-to-M transaction, $M \leq N$, for a subset of scripts in the leaf of the RPS. Since the RPS that was built in the first step can be reused, the second step can be done as many times as desired.

The advantage of RPS transactions is that even if many transactions are made using the same scheme, unlocking all those transactions can be done in one transaction with very little overhead.

The disadvantage, in comparing with percentage tree transactions, is that receivers of transactions must hold the full Merkle tree, as opposed to only their relevant Merkle proof in percentage tree transactions.

5.1 Creating an RPS transaction

The creation of an RPS is similar to that of percentage tree transaction. The script is given below.

Script 5 ScriptRPS(root)

```
state variable:
    Root = root //initial value
    Amounts = Array() //dynamic array
    Percentages = Array() //dynamic array
    PaymentBits = Array() //dynamic array
function 1toMTx(percentage, Nbit)
function unlockTx(MerkleProof, unlockScript, percentage, [unlockScriptParams])
```

The function $1toMTx$ is given in Script 7, and the function $unlockTx$ is given in Script 9.

The precise protocol for creating reusable schemes is given in the Protocol 2.

Protocol 6 creating RPS

1: **Scenario:** Alice creates an RPS with scripts $S_1, \ldots, S_N$ and percentages $P_1, \ldots, P_n$ that can be unlocked by $B_1, \ldots, B_N$, correspondingly
2: Define leaves $L_1, \ldots, L_N$, each L_i is a tuple $\{H(S), P\}$
3: Build a Merkle tree where $L_1, \ldots, L_N$ are its leaves and r is its root
4: Sends each B_i the Merkle tree
5: broadcast
    6: $Transfer\big(fromScript, 0, ScriptRPS(r), AliceSigS\big)$

5.2 Making a 1-to-M transactions via an existing RPS

Anyone can use an existing RPS to make 1-to-M ($M \leq N$) transactions.

If Charlie wants to pay a subset of $\{B_1, \ldots, B_N\}$ for which a 1-to-N transaction already exists, he can add his payment to an existing RPS instead of creating a new one. The advantage is that this creates no overhead for $B_1, \ldots, B_N$ in the size of the unlocking transaction, and only makes the execution of the smart contract $O(1)$ longer.

Each 1-to-M transaction is submitted with an additional N-bit. The amount is distributed among the B_i's for which the i-th bit is set to 1. The distribution is in the same proportions as in the RPS. In addition to the N-bit, Charlie also submits the sum of percentages of all B_i's whose bit is set to 1.

Each B_i would accept this transaction as valid only if he has all the Merkle proofs for which the bit in the N-bit is 1, otherwise he cannot verify that the transaction is valid.

The script for the function for making a 1-to-M transaction using an existing RPS is given in Figure 7.

Script 7 1toMTx(percentage, Nbit)

```
1: if percentage<1 then
2:    length := amounts.length
3:    amounts[length + 1] := Transaction.amount
4:    percentages[length + 1] := percentage
5:    paymentBits[length + 1] := Nbit
6:    return TRUE
7: end if
```

The protocol is given below.

Protocol 8 Make 1-to-M transaction using an existing RPS

1: **situation** an RPS with script $S_1, \ldots, S_N$ that can be unlocked by $B_1, \ldots, B_N$ exists. Charlie wants to pay a sum of A_c coins to some subset $B_{C_1}, \ldots, B_{C_M}$, $M < N$ with the percentage proportions given by the RPS.
2: **input**: RPS script with arrays of length K and amount A in it.
3: let $P_c = \sum_{j=1}^{M} P_{C_j}$, where P_{C_j} is the percentage of leaf C_j
4: let B_c be a N-vector, where all and only the $C_1, \ldots, C_M$ are set to 1
5: Broadcast
6: Transfer(fromScript,A_c, $ScriptRPS.1toMTx(P_c, B_c)$, [CharlieSigs])
7: $B_{C_1}, \ldots, B_{C_M}$ accept this transaction as valid if and only if P_c and B_c are correct with respect to the Merkle tree of the RPS
8: **output**: RPS with arrays of length $K + 1$ and amount $A + A_c$

5.3 Unlock RPS transactions

When Alice created an RPS, each B_i received from Alice the complete Merkle tree that corresponds to the RPS. To unlock his coins B_i uses both the Merkle proof for leaf L_i and the data to unlock S_i.

Once the script verifies the correctness of the transaction, it does two things. Firstly, it transfers all the coins associated to leaf L_i to the desired destination that B_i supplied. Secondly, it updates the cells in the *paymentBit* array to have 0 in the i-th place.

The script for the function for unlocking a transaction is given in Figure 9.

Script 9 unlockTx(MerkleProof, unlockScript, percentage, [unlockScriptParams])

```
1: if MerkleProof ∈ Merkle tree with root == 'Root' then
2:    if {H(unlockScript), percentage} == MerkleProof.leaf then
3:       if unlockScript([unlockScriptParams]) == TRUE) then
4:          amount := 0
5:          place := index of MerkleProof.leaf
6:          length := amounts.length
7:          for j = 1, ..., length do
8:             if PaymentBits[j][place] == 1 then
9:                PaymentBits[j][place] := 0
10:               amount = amount + (percentage/percentages[j]) * amounts[j]
11:            end if
12:         end for
13:         if amount == Transaction.amount then
14:            return TRUE
15:         end if
16:      end if
17:   end if
18: end if
```

The precise protocol is given in Protocol 10.

check from here till end of section

Protocol 10 Unlocking RPS transactions

1: **scenario:** an RPS to $B_1, \ldots, B_N$ exists. B_i, for some $i \in [1, \ldots, N]$ has the Merkle tree for the RPS, and C_i coins in total were sent to B_i using the RPS. B_i wants to transfer those coins to *toScript*
2: B_i broadcasts
3: Transfer$\big(ScriptRPS.unlockTx(MerkleProof, toScript, percentage, [unlockScriptParams]), C_i,$ *toScript*, $[B_i Sigs]\big)$
4: **output:** C_i coins are associated to *toScript*

5.4 Analysis

We analyze the following scenario.

1. Alice created a *RPS* with each leaf L_i of a the tree contains script $S_1, \ldots, S_N$ and percentages $P_1, \ldots, P_N$, correspondingly. In addition, $B_1, \ldots, B_N$ have the data to unlock scripts $S_1, \ldots, S_N$, correspondingly.
2. Charlie makes T 1-to-M RPS transactions.
3. Each of $B_1, \ldots, B_N$ makes a transaction unlocking the coins associated to leaves $L_1, \ldots, L_N$ correspondingly.

The strength of RPS transactions is its flexibility in creating different scenarios using the same RPS. Hence we do not compare it to a specific classical scenario (unlike the analysis for percentage trees transactions in Section 4.4).

Recall from (1) that a transaction includes two scripts, fromScript and toScript. We state the following theorem without proof since its proof is very similar to that of Theorem 4.

Theorem 6. *The following claims are true.*

1. *Each 1-to-M RPS transaction that Charlie performs in the scenario is equivalent to regular M transactions from Charlie to the corresponding B_i.*
2. *The size of toScript is step 1 is $O(1)$, where $O(1)$ is the size of a hash of the hash function. The computational complexity of step 1 is $O(1)$.*
3. *The size of toScript in the step 2 is $N + O(1)$, where $O(1)$ is a rational number in $(0, 1]$. The computational complexity of step 2 is $O(1)$.*
4. *Let $Z_1^1, \ldots, Z_N^1$ be the sizes of scripts $S_1, \ldots, S_N$ correspondingly, and $Z_1^2, \ldots, Z_N^2$ be the size of data that B_i needs to unlock his coins from $S_1, \ldots, S_N$. Let $U_1, \ldots, U_N$ be the number of steps needed for B_i to unlock his coins from $S_1, \ldots, S_N$, correspondingly. Then the size of fromScript in step 3 is $N * \log N + \sum_{i=1}^{N} \left(Z_i^1 + Z_i^2\right)$ and the computational complexity is $T * N + O(N \log N) + \sum_{i=1}^{M} U_i$.*

6 Related work

In BIP16 [5], one of the first BIP (Bitcoin Improvement Proposal) for Bitcoin, a transaction of type "Pay to script hash" (P2SH) was defined. In such transactions the sender transfers the coin to an address, where the unlock script is a hash of a script instead of an actual script. To unlock the coins one needs to supply the script, whose hash matches the hash in the transaction. This allows "a sender to fund any arbitrary transaction, no matter how complicated, using a fixed-length 20-byte hash", making it cheap and fast for the sender to make complicated transaction. The transactions defined in this paper can be recognized as a P2SH transactions, with additional state variables and partial spending of coins – two features which do not exist in Bitcoin's P2SH.

The idea of using Merkle trees to represent complex scripts is used in MAST (Merklized Abstract Syntax Trees), where each branch of the tree represents a possible true path for a script. The similarity between MAST and this solution is superficial, since MAST Markle paths represent code, and ours represent state.

Percentage trees are somewhat similar to Sum Merkle trees which were already used in the blockchain context by Maxwell and Todd [7] for proving coin reserves, and in two Ethereum research papers, one by Peter Watts [14] and one by Hassan Abdel-Rahman [4], to create one-to-many payment channels.

7 Implementation

7.1 Ethereum

The Ethereum implementation implement 1-to-N percentage tree transaction. The complete code can be found at `github.com/paulperegud/small-1toN-transactions`.

The smart contract given here can hold many percentage tree transactions. Each time someone makes a percentage tree transaction it saves the new root in a mapping of roots, in addition to how much money is related to this root.

The contract is given in Solidity. It has two storage variables:

Script 11 Ethereum contract state variables

```
1: uint256 public counter = 0;
2: mapping (uint256 => Tree) public trees;
```

The Solidity code for adding a root is:

Script 12 addTree(bytes32 _root) public payable

```
1: uint256 newTreeId = counter;
2: trees[newTreeId] = Tree({
    3: root: _root,
    4: amount: msg.value,
    5: bits: bytes32(2**256)
6: });
7: counter = counter.add(1);
```

The Solidity code for withdrawing is:

Script 13 withdraw(uint treeId, uint _index, uint _permil, address _unlockScript, bytes32 _anchor, bytes _unlockData, bytes _proof)

```
1: require(is_unspent(_treeId, _index));
2: require(_percent <= 1000000);
3: bytes32 root = trees[_treeId].root;
4: MerkleHash = keccak256(_index, _anchor, _percent, _unlockScript);
5: bytes4 fsig = bytes4(keccak256("unlock_and_spend"));
6: uint amount = _percent.mul(trees[_treeId].amount).div(1000000);
7: require(_unlockScript.delegatecall(fsig, _anchor, _unlockData, _amount));
8: mark_as_spent(_treeId, _index);
```

In this implementation, *is_unspent* and *mark_as_spent* are two helper functions, operating on a bitmask. *_unlockScript* denotes a stateless trusted script from a closed catalog. It needs to be trusted because it has full access to all the funds in the tree.

We use existing Solidity contracts for implementation, namely:

- ByteUtils (slicing bytes)
- ECRecovery (recovering signer from the ECDSA signature)
- Merkle (checking membership proofs in Merkle trees)
- SafeMath (arithmetic operations with checks for overflows)

7.2 Bitcoin

Bitcoin's Scripts (the scripting language of Bitcoin) do not support partial spending of UTXO or state variables. In the language of Section 3, Bitcoin scripts are stateless complete scripts. Hence, it seems to be impossible to implement for Bitcoin the algorithms as they are written in this paper.

It does not seem that those two properties will be added soon, or ever, to Bitcoin. But what may be added are Bitcoin Covenants [8], an extension to Script that allows for scripts to limit how UTXO is spent. This section is a high level discussion of implementing percentage tree and RPS transactions using covenants.

The main idea is to imitate partial withdrawls with covenants. A combination of $CheckOutputVerify$ and $Pattern$ opcodes is used to recursively place restrictions on how change output can be spend.

A version of percentage tree transactions can be implemented with this method. The root, percentage and N-bit would be hardcoded into the unlock script. When B_i unlocks his share of the tree, the script forces B_i to transfer the rest of the amount to a new UTXO, where the N-bit is changed to mirror the fact that B_i withdrew his part.

We believe that a similar method can also be used to implement a version of RPS transactions, though we do not discuss it here. The scripts would be long and cumbersome, and the only advantage of such a method would be for applications that are crucially dependent on making smaller 1-to-N transactions.

8 Applications

Two examples for possible applications.

8.1 Huge payment hubs

One possible outcome of Bitcoin Lightning Network and payment channels in general, is the establishment of payment hubs. Such hubs would open channels to a large amount of nodes. If most of those channels were opened in a short funding stage of the hub, it would be a slow and expensive process for the hub owners.

The payment hub could instead be set up quickly using a 1-to-N percentage tree transaction. In this case each of the N transactions is a funding transaction of a payment channel. This makes funding the payment hub as fast and as expensive as making one smaller transaction.

If Lightning network became popular enough in the future, channels might exist for a very long time (or never be closed at all). In this case the difficulty of creating payment hubs is in funding the payment channels, and not in closing them, making 1-to-N percentage tree transactions ideal for this application.

8.2 Continuous payment of salaries

Consider the case of a company with N employees. The current standard is paying the employees' salary once a month, but the digital era may change the standard to a continuous payment of salaries. This means that once every period of time, for example, once an hour, the company pays the workers' salary for one hour of work.

Using regular blockchain transactions, the company would need to create N transactions every hour, and approximately $24N$ transactions a day and $720N$ transactions a month. This would be expensive and slow for the company.

Alternatively, the company can set an RPS for its workers. The relative sizes of salaries between workers are represented by the percentages of the leaves in the RPS. Once this is done, the company can pay once an hour the $M < N$ workers that worked in this hour (surely not all the N workers work every hour due to vacation, holidays and so on).

The workers can choose when to unlock the payments. They can do it once and hour, or once a month, with very little overhead.

References

[1] Block size limit controversy. https://en.bitcoin.it/wiki/Block_size_limit_controversy. [Online, visited September 2018].

[2] Sharding faqs. https://github.com/ethereum/wiki/wiki/Sharding-FAQs. [Online, visited September 2018].

[3] Coinnarketcap - all cryptocurrencies. https://coinmarketcap.com/all/views/all/, 2018. [Online].

[4] H. Abdel-Rahman. Scalable payment pools in solidity. https://medium.com/cardstack/scalable-payment-pools-in-solidity-d97e45fc7c5c, 2018. [Online].

[5] G. Andresen. Bip 16: Pay to script hash. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki, 2012. [Online].

[6] Coindesk. Loveable digital kittens are clogging ethereum's blockchain. https://www.coindesk.com/loveable-digital-kittens-clogging-ethereums-blockchain/, 2017. [Online].

[7] G. Maxwell and P. Todd. Proving your bitcoin reserves. https://web.archive.org/web/20150406222734/https://iwilcox.me.uk/2014/proving-bitcoin-reserves, 2014. [Online].

[8] M. Möser, I. Eyal, and E. G. Sirer. Bitcoin covenants. In *International Conference on Financial Cryptography and Data Security*, pages 126–141. Springer, 2016.

[9] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. whitepaper, 2009, 2009.

[10] NewsBTC. Bitcoins transaction fee exceeds 50 dollars as network issues remain. https://www.newsbtc.com/2017/12/22/bitcoins-average-transaction-fee-surpasses-50-network-issues-remain/, 2017. [Online].

[11] A. Poelstra. Mimblewimble. 2016.

[12] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.

[13] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[14] P. Watts. Pooled payments (scaling solution for one-to-many transactions). https://ethresear.ch/t/pooled-payments-scaling-solution-for-one-to-many-transactions/590, 2018. [Online].