ARCC - Allowable Revocable Contract Chain for Bitcoin Cash

Kuldeep Singh Grewal
kuldeepbb.grewal@gmail.com
July 10, 2021

Abstract: Transactions in Bitcoin Cash are irreversible i.e permanent, which means that once the money leaves the user's wallet and gets included in a block, it cannot be taken back. Bitcoin Cash aims to be the money for the world. But in the real world, many cases require one party to be able to control the flow of funds and still give restricted access to another party to spend that money. I propose ARCC, a Bitcoin Cash Smart Contract Chain System that allows the payer to take back the money while still allowing the payee to withdraw some funds from the contract based on restrictions of time and amount.
There can be a range of applications/mechanisms possible including but not limited to, Streaming Services, Pay as you use, Recurring payments, Milestone based payouts, Project funding, Pocket money etc.

Content

Problem

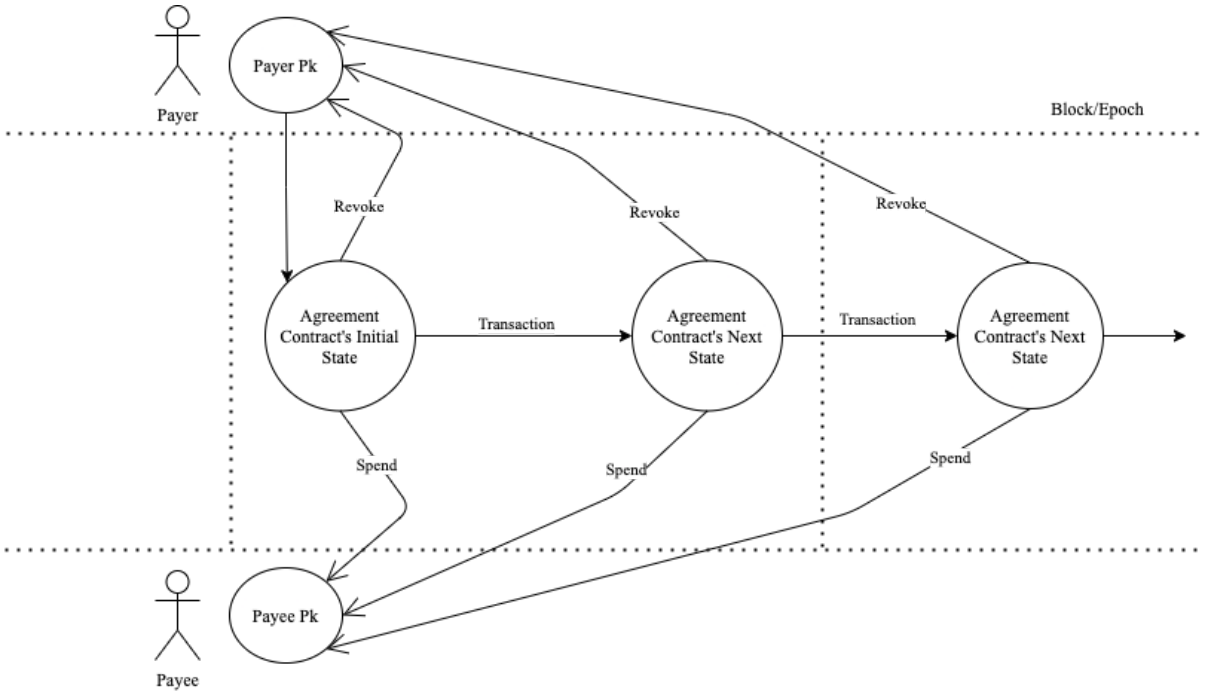
The way bitcoin works is that it allows money to be pushed to a recipient which means that once money is sent it cannot be taken back. There is no concept of refund in bitcoin. The ability to mimic a pull/revoke mechanism with spending restrictions provided by the covenants could be a potential game changer for the bitcoin cash ecosystem.

There are numerous applications that would be possible if the abilities mentioned above were provided to the users of bitcoin cash.

Solution

Overview

An ARCC system will open doors to numerous applications.



Parts

1. Agreement Contract: A smart contract written in Bitcoin Script that utilizes covenant capabilities and restricts the spending ability of payee while allowing payer to revoke access to funds. The restrictions are time and amount based.

2. Payer: The party that is responsible to fund the Agreement Contract.

3. Payee: The party that will have restricted access to spending funds.

Initial Funding Process

1. An Agreement contract is created using the following parameters: *payerPk, payeePk, maxAmountPerEpoch, epoch, remainingTime, remainingAmount, validFrom* (Discussed later in Agreement Contract Constructor Parameters). These parameters will be responsible for calculating the spending allowance of the payee.

2. The Payer is responsible for verifying the parameters used in constructing the agreement contract for the first time.

3. A cash address for Agreement contract is created using it's locking script hex.

4. The Payer can now send the money to the Agreement contract's address.

5. The Agreement contract is now funded.

Revoke
At any time during the contract's existence the payer can revoke the access of funds from the payee by simply providing their signature, signing the preimage of the transaction to transfer the amount available in the contract to any address.

Payment Process and constraints

1. The Payee of the Agreement contract has the ability to spend funds from the contract when all the following conditions are met:
    a. The amount is greater than the *dust* limit. (546 satoshi)
    b. The amount is less than *maxAmountPerEpoch*
    c. The amount is less than the *remainingAmount*. (*remainingAmount* <= *maxAmountPerEpoch*)
    d. The *remainingTime* <= *epoch*.
    e. The signatures must match to the *payeePk*.
    f. The locktime has matured.(If any)
    g. The change amount sent to the next contract state is > dust. (>546)
2. As soon as the payee makes a transaction from the contract the amount is transferred to the payee's address derived from payeePk and the remaining amount is transferred to the contract's next state.
    a. Inputs: N (Any number of inputs)
    b. Outputs:
        i. P2PKH: *amount* satisfying the condition mentioned in step above.
        ii. P2SH: The change amount(*amountToNextState*) goes to a new contract.
3. The payee will have to wait for the next epoch if the amount spent is already equal to *maxAmountPerEpoch*.
4. The payee can still make another payment if the total amount spent within the epoch is less than the *maxAmountPerEpoch* and greater than the dust limit.

All this will be ensured by the smart contract contained in the redeemScript of the UTXO.

Cases

1. Payee withdraws the complete amount.
   As soon as Payee withdraws the complete amount the following parameters get affected:

a. *validFrom*: This parameter is set to the current block height. Or locktime if any.

b. *remainingAmount*: The remaining amount is set to 0 which makes it impossible for payee to spend funds from the contract until the next epoch has started.

c. *remainingTime*: This parameter is set to *epoch* - time since last epoch.

2. Payee withdraws a partial amount:
   As soon as Payee withdraws a partial amount the following parameters get affected.
   a. *validFrom*: This parameter is set to current block height. Or locktime if any.

   b. *remainingAmount*: The remaining amount is set to *maxAmountPerEpoch - amount*

   c. *remainingTime*: This parameter is set to *epoch* - time since last epoch.

3. Payee misses the epoch and fails to withdraw.
   The Payee will only be able to withdraw any amount from the current epoch time frame. After the time has passed, the user cannot withdraw the amount from past missed epochs. The misses can easily be prevented by adding a layer to the current contract.

4. Payer revokes the access to money:
   As soon as the payer makes a revoke type transaction, the money from the Agreement contract gets transferred back to the Payer's prefered address.

5. The Epoch parameter is set to 0. (See next section)

Slots

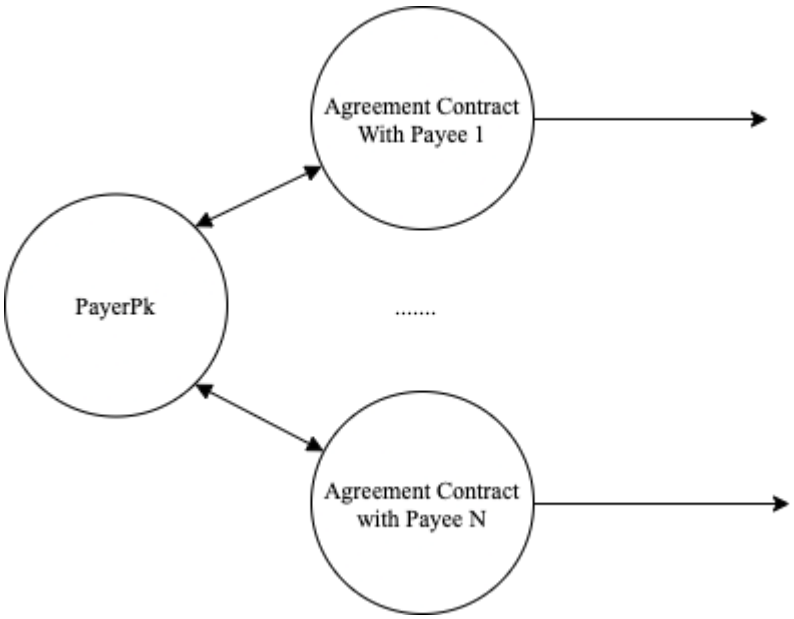
For the special case of epoch = 0, a concept of slot could be introduced as a layer on top of the contract. A slot may represent time, work, or anything measurable. Since epoch is 0, the contract is only bound to *amount* and *maxAmountPerEpoch*. The Payee can make multiple transactions immediately to fetch all the balance in the contract but if the concept of slot is introduced then the payer can periodically add funds to the contract which payee can immediately spend. This can reduce the limitation of 1 epoch being equal to 1 block height i.e ~10 mins. Now both parties will be able to exploit the 0-conf feature of Bitcoin Cash and hence make as many transactions as they wish per epoch or block.

Refill Process

The contract can easily be refilled by sending more funds to the latest address of the Agreement contract. Everyone has the ability to add funds to the Agreement contract but only Payer or Payee are allowed to spend the money, hence, if a third party sends funds to the contract then they cannot get it back.

Relation

The Payer/Payee can be associated with any number of Agreement contracts.



Smart Contract

Agreement Contract Constructor Parameters

1. *payerPK:* Public Key of the Payer.

2. *payeePk:* Public Key of the Payee.

3. *epoch:* Timeframe(in blocks, 1 epoch = ~10 min/block) that is responsible for restricting the spending ability of payee by only allowing it to spend up to *maxAmountPerEpoch* in any number of transactions within the epoch time frame.
   Note: A valid epoch must always be >= 0. For cases where epoch is 0, the contract is not bound by time but only by amount.

4. *maxAmountPerEpoch:* Maximum amount spendable by the payee per epoch.

5. *remainingTime:* The time left before the next epoch starts.
   Note: The Payer **must** make sure that the *remainingTime* = *epoch* before constructing the Agreement contract. Although it's effects are not severe, the first epoch may be shorter than mentioned in epoch parameter. Since the calculation of *newRemainingTime* is done by calculating the *passedTime* followed by the modulo(%) operation.

6. *remainingAmount:* Remaining spendable amount left for payee before the next epoch starts.

7. *validFrom*: Timelock of the contract. The Payer **must** make sure that *validFrom* = *locktime* before funding the Agreement Contract.

The state of the contract resets after each epoch except the *validFrom* parameter.

It is important to note here that along with the first parameter i.e *payerPK* a second parameter *payerContractScriptHash* could be introduced which has *payerPK* as it's owner. This can create a new form of Contract Chain based system. This paper does **not** describe this concept in more detail.

Example: A => B => C => .... XYZ. Each having their own *epoch* and *maxAmountPerEpoch.*

Deriving the next state.

The contract trims the bytecode/scriptcode by first 15 bytes keeps the right half and then pushes the new state variables *locktime*, *remainingAmount* and *remainingTime* each of 4 bytes and 1 byte each for push opcode to the front of the right half hence re-creating a new state script code.

The next state of the contract i.e address and locking script is derived by providing the exact parameters to the contract constructor which were used in recreating the script code in the step above.

For example, If all the previous parameters are same except *remainingAmount* then while constructing the new instance of the contract only the new *remainingAmount* variable should be changed while others remain the same.

State 0:
- *maxAmountPerEpoch*: 3000
- *remainingAmount*: 3000

Transaction:
- *amount*: 1000
- *remainingAmount*: 3000-1000 = 2000

State 1:
- *maxAmountPerEpoch*: 3000
- *remainingAmount*: 2000

prevContract = [*payerPk, payeePk, maxAmountPerEpoch, epoch, remainingTime, remainingAmount, validFrom*]

nextContract = [*payerPk, payeePk, maxAmountPerEpoch, epoch, remainingTime,* ***newRemainingAmount****, validFrom*]

Note: Each transaction has two outputs and the second output is always sent to a new state.

References

[1] Tobias Ruck ‘BeCash’ https://be.cash/becash.pdf
[2] Rosco kalis ‘CashScript’ https://kalis.me/uploads/msc-thesis.pdf