ARCC - Allowable Revocable Contract Chain System 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. 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.

Existing Solutions and Comparison.

CashChannels: CashChannels lets users per-approve future transactions, valued in any currency, to another person or company.

1. Payment Authorizations:
    a. CashChannels: The spender sends the authorization message to the receiver, who holds it until the payment time. At or after the payment time, the receiver can use the authorization to create a single transaction from the channel for the correct amount.
    b. ARCC: There is no concept of Payment Authorization, the only way the payee can keep on getting the payments is simply by deriving the next state. (More details in the sections below)

2. Channel Identifier and Payment Number:
    a. CashChannels: Channel Identifier and Payment Number ensures that the authorization can only be used once for the correct channel. The payment number is used as an identifier for an authorization and this number should be increased by one for each authorization signed.
    b. ARCC: There is no concept of Channel Identifier and Payment Number. The current contract state is derived from the last state and time. If a previous state still has funds then the payer can easily pull it out to any address. Moreover, a payee can make as many transactions as they wish during an epoch to redeem the funds allotted. (More details in the sections below)

3. Epoch:
    a. CashChannels: There is no concept of epoch.
    b. ARCC: Epoch acts as a window of time. (epoch = 1 = ~10 min/block) (More details in the sections below.)

4. Oracles:
    a. CashChannels: Cash Channels rely on Oracles to send a message in order to come to an agreed price and hence make a transaction.(if currency is not BCH)
    b. ARCC: An ARCC system does not depend on oracle and leaves the implementation details on the layer on top of the contract(For example: An aggregated price feed timestamp server). If the price changes significantly then a new contract initial state can be created i.e contract fork, by adjusting the values of *maxAmountPerEpoch* and *epoch.* (more details in section: Dealing with price volatility)

       An ARCC system may or may not implement the Oracle based system depending upon how it's implemented. If there is a need for a third party like Oracle to be involved in the system then it can be integrated by providing the public key of the oracle at the beginning of the contract creation and later at the time when payee is spending the transaction, they must provide a signed message consisting the spendable amount in satoshis from the oracle and then the contract can verify the signatures using the OP_CHECKDATASIG, verifying that the signer of the message containing the amount in indeed the Oracle. The only condition after this which needs to be satisfied is that the spendable amount mentioned must be less than the remaining amount allowed per epoch.

5. Payment Time:
    a. CashChannels: The time at which the authorization may be used by the receiver.

b. ARCC: *validFrom* could be closely treated as to what *Payment Time* is on CashChannels. *validFrom* represents the time constraint on the Payee. (More details in sections below)

Solution

Overview

An ARCC system will open doors to numerous applications.

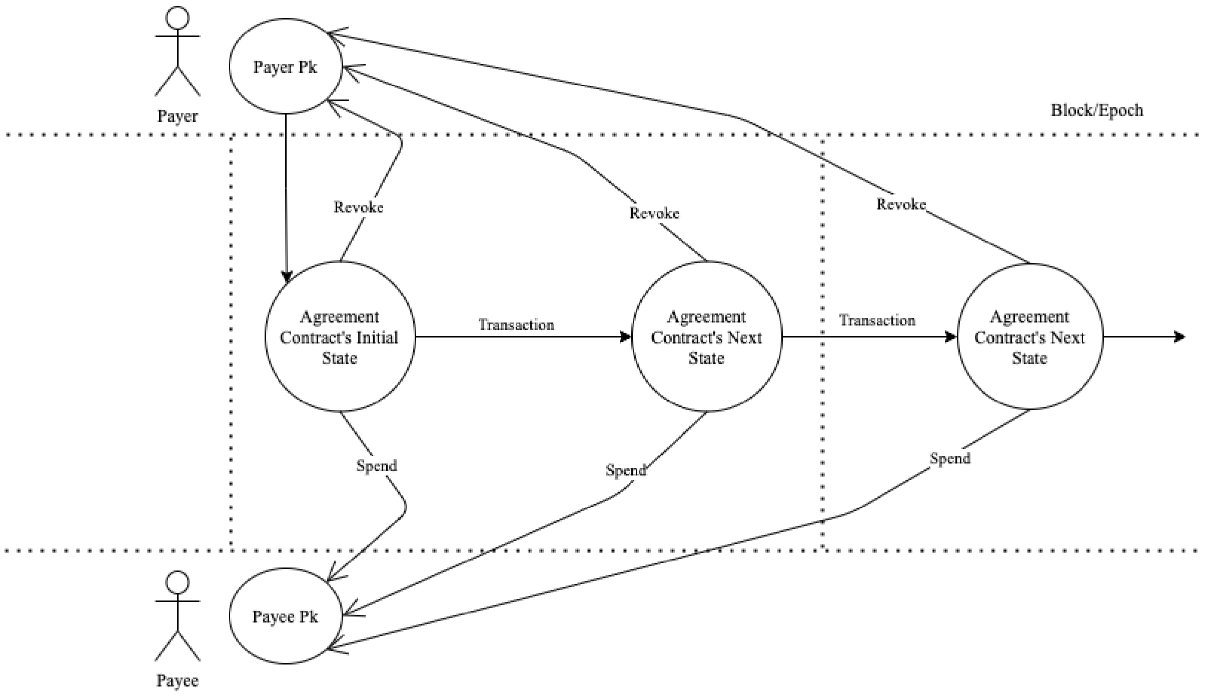
Payer
Payer Pk
Block/Epoch
Revoke
Revoke
Revoke
Agreement Contract's Initial State
Transaction
Agreement Contract's Next State
Transaction
Agreement Contract's Next State
Spend
Spend
Spend
Payee Pk
Payee

Figure 1: Visualization of an ARCC system.

Parts

1. Agreement Contract: A smart contract written in Bitcoin Script that restricts the spending ability of payee while allowing payer to revoke access to funds at any time. 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, epoch, maxAmountPerEpoch, 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 CashAddress for Agreement contract is created using it's locking script.

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, and transfering the amount available in the contract to any address.

Constraints

1. The Payee of the Agreement contract has the ability to spend funds from it when **all** the following conditions are met:

    a. The amount being spent is greater than the *dust* limit. (dust = 546 satoshis)

    b. The amount being spent is less than *maxAmountPerEpoch*

    c. *maxAmountPerEpoch* >= *dust* (dust = 546)

    d. The amount being spent is also less than the *remainingAmount*. (*remainingAmount* <= *maxAmountPerEpoch*)

    e. The *remainingTime* <= *epoch* && >= 0

    f. The *newRemainingAmount* >= 0 && < = *maxAmountPerEpoch*

    g. The signatures must match to the *payeePk*.

    h. The locktime has matured.

    i. The change amount sent to the next contract state is > dust. (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 the next contract state.

3. The payee will have to wait for the next epoch if the amount spent is already equal to *maxAmountPerEpoch*.

4. If *remainingAmout* < 546, then the payee will have to wait for the next epoch in order to start making payments.

5. 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 and network rules.

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. (Ignores the missed epochs)

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* or previous *remainingAmount - amount.*

   c. Change amount is set to *amountToNextState.*

   d. *remainingTime*: This parameter is set to *epoch* - time since last epoch. (Ignores the missed epochs)

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 missed epochs. The misses can easily be prevented by adding a layer to the current contract.

   Note, this does not lock or burn money, the previously missed epochs are simply ignored.

4. Payer revokes the access to money:
   As soon as the payer makes a revoke 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 by *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.

Example: 1 Epoch = 2 Blocks, 1 Block = 4 Slots.

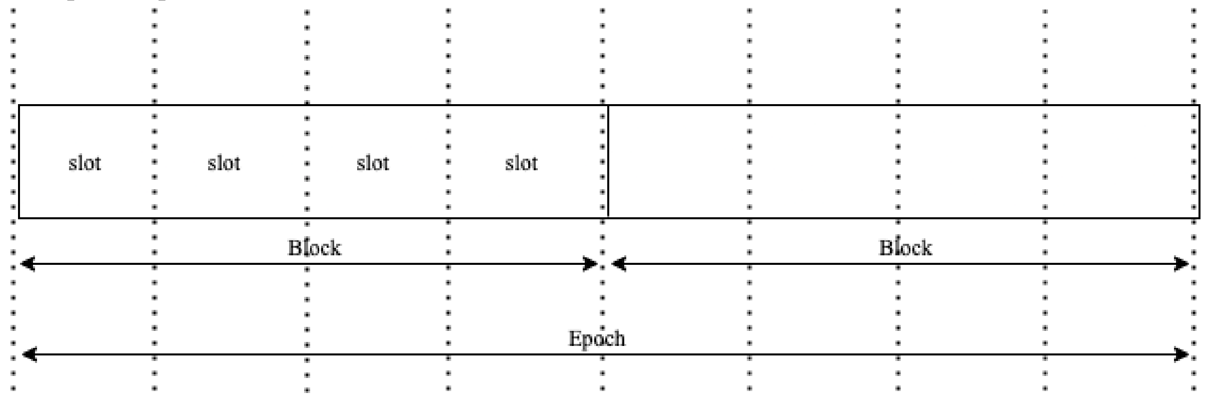

Figure 2: Visualisation of Epochs, Blocks and Slots.

Refill Process

The contract can easily be refilled by sending more funds to the address of the current Agreement contract's state. 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.

Relations

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

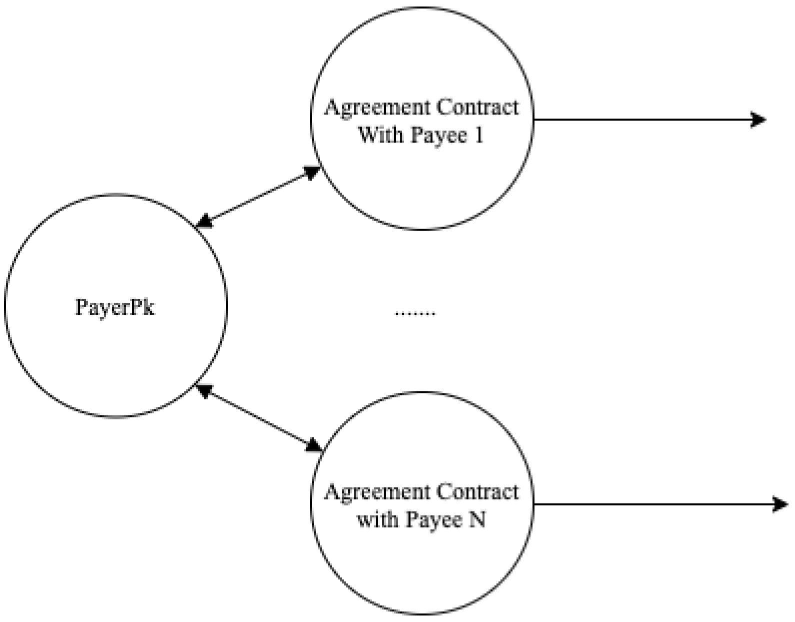
Agreement Contract With Payee 1
PayerPk
....
Agreement Contract with Payee N

Figure 3: Visualization of Agreement Relations of Payer with other Payees

Dealing with price volatility

When a price fluctuation happens with respect to any other real world asset or currency at the time which is close to the start of the next epoch, a new Agreement contract could be set by both the parties agreeing to the new(mostly the same) parameters.

- Changes to *maxAmountPerEpoch* are inversely proportional to the price.
- Changes to *epoch* are directly proportional to the price.

These changes will lead to a new contract being created.

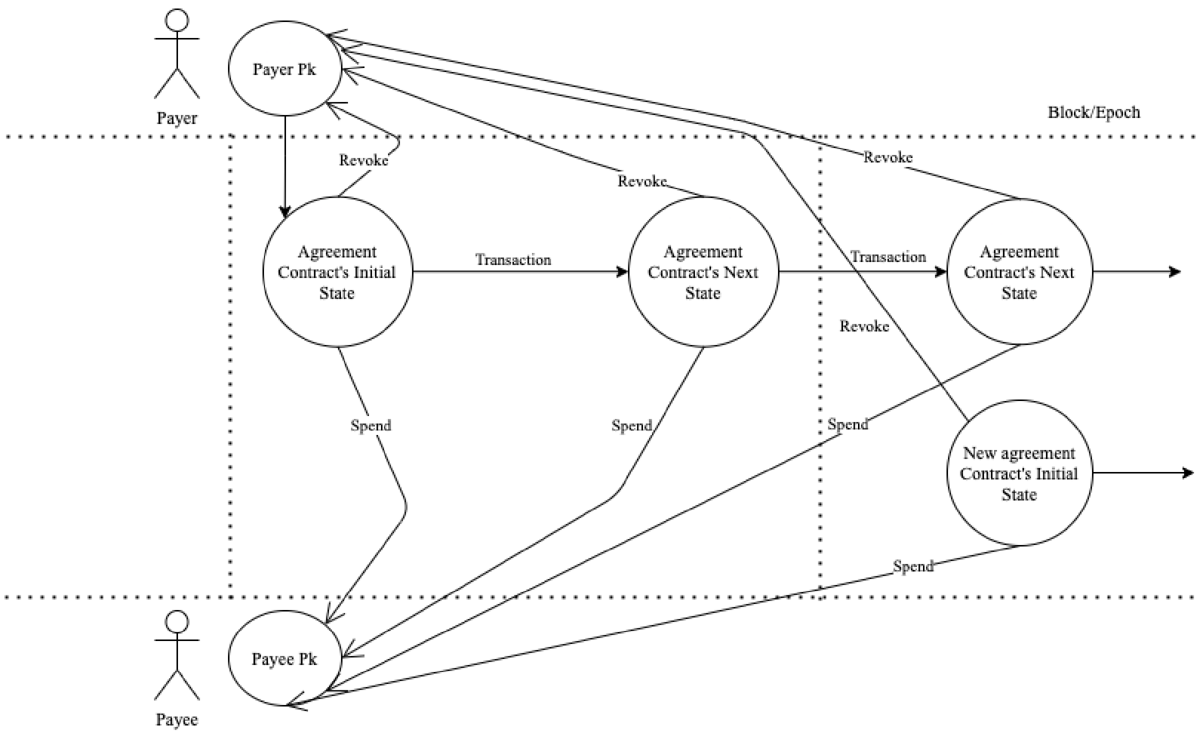
Payer
Payer Pk
Block/Epoch
Revoke
Revoke
Revoke
Agreement Contract's Initial State
Transaction
Agreement Contract's Next State
Transaction
Agreement Contract's Next State
Revoke
Spend
Spend
Spend
New agreement Contract's Initial State
Spend
Payee
Payee Pk

Figure 4: Visualization of Method-1 and Method-2 under Dealing with Price Volatility.

1. **Method-1, Contract forking**: Change *maxAmountPerEpoch*

Changing the *maxAmountPerEpoch* would simply mean that the payee can only spend the new amount mentioned in the contract.

If the price goes down then *maxAmountPerEpoch* should be increased accordingly.
If the price goes up then *maxAmountPerEpoch* should be decreased accordingly.

2. **Method-2, Contract forking**: Change *epoch*

Changing the *epoch* would simply mean that the payee can only spend the amount restricted by the new time frame.

If the price goes down then *epoch* should be decreased accordingly.
If the price goes up then the *epoch* should be increased accordingly.

3. **Method-3**: epoch = 0

In case of Epoch = 0, There is a need to establish a new contract **only** when *maxAmountPerEpoch* needs to be changed as the contract would only be bound by *amount* and *maxAmountPerEpoch* but, if the concept of slots is introduced, then the payer can simply increase or decrease the number of slots and/or *maxAmountPerSlot* to make sure the payout is maintained.

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, epoch = 1 = ~10 min/block) that is responsible for restricting the spending ability of payee by only allowing 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 is recommended to make sure that the *remainingTime* = *epoch* before funding the initial Agreement contract. Although, it’s effects are not severe, the first time frame window may be shorter than what was mentioned in epoch parameter.

6. *remainingAmount:* Remaining spendable amount left for payee before the next epoch starts. The Payer is recommended to make sure that the remainingAmount = maxAmountPerEcoch before funding the initial Agreement contract.

7. *validFrom*: Block Height/ Timelock of the 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 may have *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.*

Spend

Payee must provide the following input parameters when spending from the contract.

1. *payeeSig*: Signature of the Payee.
2. *amountToNextState*: Amount to be sent to the next state.
3. *amount*: Amount being spent by the payee.

Revoke

Payer must provide the following input parameters when spending from the contract.

1. *payerSig*: Signature of the Payer.

Deriving the next state.

Each transaction is responsible for creating a new state, in order to spend from the current state, the payee must derive the next state's P2SH address. This can be done by either calculating or predicting the constructor parameters of the next state. Each transaction must have two outputs with the first output being sent to the payee's address and the second output to the next state's P2SH address.

Each state can also act as an initial state and there is no need to verify the previous contract chain. With correct parameters, anyone can derive the contract's state and the script's address that has the balance but the locked money can only be spent by either Payer or Payee.

The contract uses OP_CHECKDATASIGVERIFY and OP_CHECKSIGVERIFY to be able to introspect the transaction spending it.

It uses OP_SPLIT and OP_SIZE(Counting the bytes length of the preimage) on the preimage from the end to fetch the *hashOutputs* and *nlocktime*. It requires that the preimage be put in the ScriptSig.

It needs this ability to do the following:

- Restrict the spending of the amount only to payee and newly created contract state, by verifying that the *hashOutputs's* (32-bytes hash) preimage in the ScriptSig matches the hash.
- *nlocktime* to calculate the passedTime and to push it's value as the first 4 bytes in the scriptCode i.e. validFrom of the new contract state.

The contract uses some of the OPcodes of the Bitcoin Script such as OP_GREATERTHANEQUAL, OP_WITHIN, OP_MOD and OP_VERIFY, to enforce the restrictions on *remainingAmount* and *remainingTime* at the time of spending.

The contract trims the scriptCode by first 15 bytes, keeps the right half and then pushes the new state variables n*locktime*(4 bytes), *remainingAmount*(4 bytes) and *remainingTime*(4 bytes)*,* (0x04)1 byte each to push data to the front hence re-creating a next state script code.

References

[1] Tobias Ruck ‘BeCash’ https://be.cash/becash.pdf, 2019
[2] Jason Dreyzehner, ‘CashChannels’,
https://blog.bitjson.com/cashchannels-recurring-payments-for-bitcoin-cash/
[3] Rosco kalis ‘CashScript’ https://kalis.me/uploads/msc-thesis.pdf , 2019
[4] Tendo Pein ‘BCH Covenants with spedn’
https://read.cash/@pein/bch-covenants-with-spedn-c1170a02, 2020