BeCash – a secure sender-offline payment system for Bitcoin Cash

Tobias Ruck

October 23, 2019

Abstract. Using Bitcoin as payment system for in-person commerce has many drawbacks, as both payer and payee have to be online to perform the transaction. We propose a solution that allows payments to be sent securely and reliably from an offline wallet to an online wallet. After querying a UTXO lookup service, payees can redeem the coins immediately. This could have a range of applications, including, but not limited to, Point-Of-Sales systems, ATMs, paid-access faregates, on-site gambling etc.

1 Problem

In theory, sending transactions offline with Bitcoin is possible out-of-the box. Let's consider an offline Payer and an online Payee. A simple but flawed version of offline transactions, using QR codes to communicate, would look like this:

1. While online, the wallet syncs the internal UTXO set of the wallet.
2. Offline, Payer scans the address and payment amount encoded in a QR code.
3. The wallet creates a transaction sending the requested amount to the address. This transaction is based on the internal UTXO set. Assuming the merchant broadcasts the transaction, the wallet then removes spent UTXOs and adds the change UTXOs.
4. The wallet displays a QR code of the transaction.
5. Payee scans the transaction, and broadcasts it on the network.
6. If broadcasting succeeded, the payment is completed.

1.1 Drawbacks

There are three main drawbacks of this approach, compared to a fully offline solution.

First, if the merchant fails to broadcast the transaction, the wallet will update the UTXO set erroneously, which means Payee must either undo this transaction manually in their wallet, or re-sync their wallet using the Internet.

Second, receiving money offline requires either sending the received transaction via QR codes or re-syncing the wallet online.

Third, while even older phones are certainly capable of performing this scheme, smart cards have much less processing power and less space to perform complicated transaction building.

Fourth, at least some online communication might be unavoidable, which makes deployment on offline devices such as smart cards technically challenging.

While some of these issues can be mitigated somewhat, they come at the cost of either a disproportionately increased complexity, and/or a degradation of the user experience.

2 Solution

2.1 Overview

A fully offline system would have none of these problems and is described below.

2.1.1 Parts

1. A smart contract written in Bitcoin Script that utilizes covenant capabilities of `OP_CHECKDATASIG` and `OP_CHECKSIG`.
2. A UTXO indexer service, which enables a lookup of Payee's current UTXO given their public key.
3. An offline wallet which signs incoming payment requests.
4. An online wallet which can be used to both redeem signed payment requests and to fund/refill the offline wallet non-interactively.
5. A Point-Of-Sale system, which verifies signed payment requests and broadcasts them on the network.

2.1.2 Initial funding

To be able to send funds to other parties, the offline wallet has to be funded first. This is done using an already funded online wallet.

1. Payer generates a random 256-bit secret and stores it on the offline wallet device, along with a nonce which is set to `MIN_INT`.
2. The offline wallet derives a public key using the secp256k1 elliptic curve and sends it along with the nonce to an online wallet.

3. The online wallet instantiates the smart contract using the public key and nonce from the offline wallet, hashes it into a P2SH address and sends some initial funds to this address.

4. The offline wallet is now funded.

2.1.3 Payment process

Given a properly funded offline wallet, it can be used to pay an online Payee.

1. Payee creates a payment request which contains $hash160(payeePk)$ and $paymentAmount$, and sends it to the offline wallet.

2. The offline wallet signs the following hash using its secret and sends the signature to Payee, along with the wallet's public key and nonce: $sha256(hash160(payeePk)||paymentAmount||newNonce)$

3. Payee verifies the following, in order:
   a) They calculate the hash as specified above and verify the signature using the secp256k1 elliptic curve.
   b) They query the UTXO service and verify a UTXO with the wallet's public key exists,
   c) that it has sufficient balance
   d) and that the nonce of the UTXO is strictly less than the nonce provided by the offline wallet.

4. The payment can now be displayed as successful, now the payment can be broadcast.

5. Payee creates a transaction with one input and two (or possibly more) outputs. For this, Payee instantiates the smart contract twice, once using the current nonce obtained from the UTXO lookup and once using the nonce provided by the offline wallet. To create the aforementioned input, the first instantiation of the contract is used as redeemScript. The scriptSig feeding the parameters to the smart contract is created as outlined below. Then, a P2SH output is added which sends the leftover money to the customer using the second instantiation of the contract as P2SH address. Finally, they add one or more P2PKH outputs which send the paid money toPayee, sign the transaction and broadcast it to the network.

6. In case Payer performed a double-spend, the UTXO lookup service forwards a double-spend proof, and Payee can either immediately abort/revert the economic transaction associated with the fraudulent payment or automatically report Payer to appropriate authorities.

2.1.4 Refilling process

Refilling is the same as the payment process, only that the payment amount is negative (i.e. increasing the amount of the wallet) and that no signature is required from the offline wallet. This is an implementation detail and not further expanded on in this document.

2.2 Transaction layout

Transactions of this scheme have the UTXO of the offline wallet as only input. Additional inputs are allowed but usually unnecessary, as the offline wallet's value is sufficient for transaction fees. This input contains all the data that proves that this transaction only redeems as many coins from the offline wallet as the Payer's signature allows it to redeem. If not, the transaction will be invalid by the terms of the smart contract and the coins cannot be redeemed.

The first output must always be the leftover change output (except when its amount falls below the dust limit). This output's P2SH address must be the hash of the original scriptCode of the UTXO, with the modifications as outlined below.

The outputs that send *paymentAmount* to Payee must come after the leftover change output.

All of this will be ensured by the smart contract contained in the redeemScript of the UTXO, as explained in the next section.

2.3 Smart contract

2.3.1 Contract state machine

The contract is a state machine, where state transitions are enforced by the contract itself. Depending on the inputs and current state, a different state transition occurs and the contract will have a different new state.

A state transition occurs with each transaction.

The current state is encoded in the redeemScript, in the very first operation, as a PUSHDATA operation. After this operation, the state is the top stack item and the contract can use this state data and the given inputs which are the remaining stack items to compute the next state. Then, the contract ensures that the P2SH address for the newly created UTXO hashes to the same redeemScript as that of the currently spent UTXO, except that the first pushop is replaced with the new state. This requires removing the first N bytes from the scriptCode (where N is the state size in bytes) and prepending the new state to the trimmed scriptCode.

In the contract outlined here, the state is the current nonce of the wallet, encoded as 64-bit number (i.e. the result of applying `8 OP_NUM2BIN`). Hence—as the opcode for pushing 8 bytes also requires 1 byte—$N = 9$.

2.3.2 High-level script constructor parameters

The script has two constructor parameters, oldNonce and walletPk, where oldNonce is the current state of the contract and thus the very first pushop of the contract. walletPk determines the owner of the contract.

2.3.3 Script Inputs

The redeemScript receives the following inputs. Some inputs, e.g. parts of the preimage, are omitted for clarity.

1. $newNonce$. The nonce signed by $walletSig$.
2. $oldValue$. The value of the UTXO in satoshis.
3. $paymentAmount$. The amount to be withdrawn from the wallet. Signed by $walletSig$.
4. $walletSig$. Signature for $sha256(hash160(payeePk)||paymentAmount||newNonce)$, signed by $walletPk$.
5. $payeePk$. The public key of Payee. The address of the public key is signed by $walletSig$.
6. $payeeSig$. Signature signing the preimage for this transaction, signed by $payeePk$.
7. $payeeOutputs$. The outputs Payee wants the payment to be sent to.
8. $lokadId$. Fixed 4-byte constant. Allows the transaction to be found by an indexer.

2.3.4 Contract constraints

The contract ensures the following high-level constraints:

1. $newNonce > oldNonce$.
2. $sha256(hash160(payeePk)||paymentAmount||newNonce)$ is signed by $walletPk$ with $walletSig$.
3. The transaction is signed by $payeePk$ with $payeeSig$.
4. For the first output:
   a) Its script is P2SH.
   b) Its address is the same as of the current UTXO, only that the state variable is now $newNonce$.
   c) Its value is $oldValue - paymentAmount$.
5. Additional outputs are $payeeOutputs$.