```
BIP: 78
Layer: Applications
Title: A Simple Payjoin Proposal
Author: Nicolas Dorier <nicolas.dorier@gmail.com>
Replaces: 79
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0078
Status: Draft
Type: Standards Track
Created: 2019-05-01
License: BSD-2-Clause
```

Introduction

Abstract

This document proposes a protocol for two parties to negotiate a coinjoin transaction during a payment between them.

Copyright

Motivation

When two parties (later referred to as sender and receiver) want to transact, most of the time, the sender creates a transaction spending their own Unspent Transaction Outputs (UTXOs), signs it and broadcasts it on the network.

This simple model gave birth to several heuristics impacting the privacy of the parties and of the network as a whole.

- Common input ownership heuristic: In most transactions, all the inputs belong to the same party.
- Change identification from scriptPubKey type: If all inputs are spending UTXOs of a certain scriptPubKey type, then the change output is likely to have the same scriptPubKey type, too.
- Change identification from round amount: If an output in the transaction has a round amount, it is likely an output belonging to the receiver.

We will designate these three heuristics as `common-input`, `change-scriptpubkey`, `change-round-amount`.

The problems we aim to solve are:

- For the receiver, there is a missed opportunity to consolidate their own UTXOs or making payment in the sender's transaction.
- For the sender, there are privacy leaks regarding their wallet that happen when someone applies the heuristics detailed above to their transaction.

Our proposal gives an opportunity for the receiver to consolidate their UTXOs while also batching their own payments, without creating a new transaction. (Saving fees in the process) For the sender, it allows them to invalidate the three heuristics above. With the receiver's involvement, the heuristics can even be poisoned. (ie, using the heuristics to intentionally mislead blockchain analysis)

Note that the existence of this proposal is also improving the privacy of parties who are not using it by making the three heuristics unreliable to the network as a whole.

Relation to BIP79 (Bustapay)

Another implementation proposal has been written: BIP79 Bustapay.

We decided to deviate from it for several reasons:

- It was not using PSBT, so if the receiver wanted to bump the fee, they would need the full UTXO set.
- Inability to change the payment output to match scriptPubKey type.
- Lack of basic versioning negotiation if the protocol evolves.
- No standardization of error condition for proper feedback to the sender.

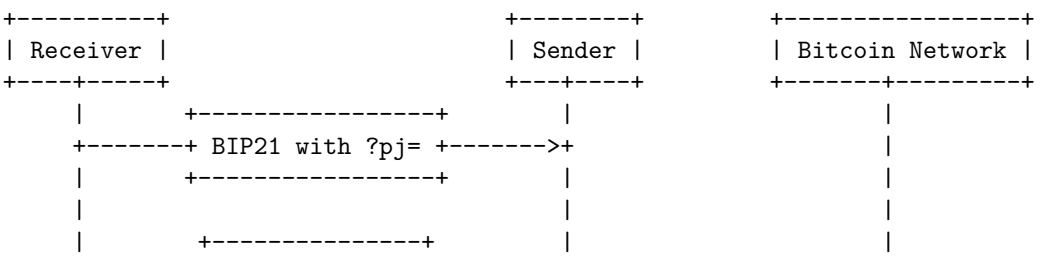
Other than that, our proposal is very similar.

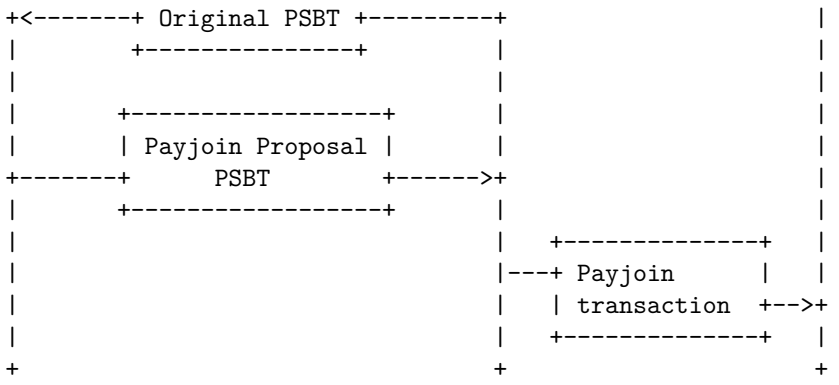
Specification

Protocol

In a payjoin payment, the following steps happen:

- The receiver of the payment, presents a BIP 21 URI to the sender with a parameter `pj=` describing a payjoin endpoint.
- The sender creates a signed, finalized PSBT with witness UTXO or previous transactions of the inputs. We call this PSBT the `original`.
- The receiver replies back with a signed PSBT containing his own signed inputs/outputs and those of the sender. We call this PSBT `Payjoin proposal`.
- The sender verifies the proposal, re-signs his inputs and broadcasts the transaction to the Bitcoin network. We call this transaction `Payjoin transaction`.





The original PSBT is sent in the HTTP POST request body, base64 serialized, with `text/plain` in the `Content-Type` HTTP header and `Content-Length` set correctly. The payjoin proposal PSBT is sent in the HTTP response body, base64 serialized with HTTP code 200.

To ensure compatibility with web-wallets and browser-based-tools, all responses (including errors) must contain the HTTP header `Access-Control-Allow-Origin: *`.

The sender must ensure that the url refers to a scheme or protocol using authenticated encryption, for example TLS with certificate validation, or a .onion link to a hidden service whose public key identifier has already been communicated via a TLS connection. Senders SHOULD NOT accept a url representing an unencrypted or unauthenticated connection.

The original PSBT MUST:

- Have all the `witnessUTXO` or `nonWitnessUTXO` information filled in.
- Be finalized.
- Not include fields unneeded for the receiver such as global xpubs or keypath information.
- Be broadcastable.

The original PSBT MAY:

- Have outputs unrelated to the payment for batching purpose.

The payjoin proposal MUST:

- Use all the inputs from the original PSBT.
- Use all the outputs which do not belongs to the receiver from the original PSBT.
- Only finalize the inputs added by the receiver. (Referred later as `additional inputs`)
- Only fill the `witnessUTXO` or `nonWitnessUTXO` for the additional inputs.

The payjoin proposal MAY:

- Add, remove or modify the outputs belonging to the receiver.

The payjoin proposal MUST NOT:

- Shuffle the order of inputs or outputs, the additional outputs or additional inputs must be inserted at a random index.
- Decrease the absolute fee of the original transaction.

BIP21 payjoin parameters

This proposal is defining the following new BIP 21 URI parameters:

- `pj=`: Represents an http(s) endpoint which the sender can POST the original PSBT.
- `pjos=0`: Signal to the sender that they MUST disallow payment output substitution. (See Unsecured payjoin server)

Optional parameters

When the payjoin sender posts the original PSBT to the receiver, he can optionally specify the following HTTP query string parameters:

- `v=`, the version number of the payjoin protocol that the sender is using. The current version is `1`.

This can be used in the future so the receiver can reject a payjoin if the sender is using a version which is not supported via an error HTTP 400, `version-unsupported`. If not specified, the receiver will assume the sender is `v=1`.

If the receiver does not support the version of the sender, they should send an error with the list of supported versions:

```
{
    "errorCode": "version-unsupported",
    "supported" : [ 2, 3, 4 ],
    "message": "The version is not supported anymore"
}
```

- `additionalfeeoutputindex=`, if the sender is willing to pay for increased fee, this indicate output can have its value substracted to pay for it.

If the `additionalfeeoutputindex` is out of bounds or pointing to the payment output meant for the receiver, the receiver should ignore the parameter. See fee output for more information.

- `maxadditionalfeecontribution=`, if the sender is willing to pay for increased fee, an integer defining the maximum amount in satoshis that the sender is willing to contribute towards fees for the additional inputs. `maxadditionalfeecontribution` must be ignored if set to less than zero. See fee output for more information.

Note that both `maxadditionalfeecontribution=` and `additionalfeeoutputindex=` must be specified and valid for the receiver to be allowed to decrease an output belonging to the sender. This fee contribution can't be used to pay for anything else than additional input's weight.

- `minfeerate=`, a decimal in satoshi per vbyte that the sender can use to constraint the receiver to not drop the minimum fee rate too much.
- `disableoutputsubstitution=`, a boolean indicating if the sender forbids the receiver to substitute the receiver's output, see payment output substitution. (default to `false`)

Receiver's well known errors

If for some reason the receiver is unable to create a payjoin proposal, it will reply with a HTTP code different than 200. The receiver is not constrained to specific set of errors, some are specified in this proposal.

The errors have the following format:

```
{
    "errorCode": "leaking-data",
    "message": "Key path information or GlobalXPubs should not be included in the original P
}
```

The well-known error codes are:

Error code	Meaning
unavailable	The payjoin endpoint is not available for now.
not-enough-money	The receiver added some inputs but could not bump the fee of the payjoin proposal.
version-unsupported	This version of payjoin is not supported.
original-psbt-rejected	The receiver rejected the original PSBT.

The receiver is allowed to return implementation specific errors which may assist the sender to diagnose any issue.

However, it is important that error codes that are not well-known and that the message do not appear on the sender's software user interface. Such error codes or messages could be used maliciously to phish a non technical user. Instead those errors or messages can only appear in debug logs.

It is advised to hard code the description of the well known error codes into the sender's software.

Fee output

In some situation, the sender might want to pay some additional fee in the payjoin proposal. If such is the case, the sender must use both optional param-

eters `additionalfeeoutputindex=` and `maxadditionalfeecontribution=` to indicate which output and how much the receiver can substract fee.

There is several cases where a fee output is useful:

- The sender's original transaction's fee rate is at the minimum accepted by the network, aka `minimum relay transaction fee rate`, which is typically 1 satoshi per vbyte.

In such case, the receiver will need to increase the fee of the transaction after adding his own inputs to not drop below the minimum relay transaction fee rate.

- The sender's wallet software is using round fee rate.

If the sender's fee rate is always round, then a blockchain analyst can easily spot the transactions of the sender involving payjoin by checking if, when removing a single input to the suspected payjoin transaction, the resulting fee rate is round. To prevent this, the sender can agree to pay more fee so the receiver make sure that the payjoin transaction fee is also round.

- The sender's transaction is time sensitive.

When a sender pick a specific fee rate, the sender expects the transaction to be confirmed after a specific amount of time. But if the receiver adds an input without bumping the fee of the transaction, the payjoin transaction fee rate will be lower, and thus, longer to confirm.

Our recommendation for `maxadditionalfeecontribution=` is `originalPSBTFeeRate * vsize(sender_input_type)`.

sender_input_type	vsize(sender_input_type)
P2WPKH	68
P2PKH	148
P2SH-P2WPKH	91
P2TR	58

Receiver's original PSBT checklist

The receiver needs to do some check on the original PSBT before proceeding:

- Non-interactive receivers (like a payment processor) need to check that the original PSBT is broadcastable. `*`
- If the sender included inputs in the original PSBT owned by the receiver, the receiver must either return error `original-psbt-rejected` or make sure they do not sign those inputs in the payjoin proposal.
- If the sender's inputs are all from the same scriptPubKey type, the receiver must match the same type. If the receiver can't match the type, they must return error `unavailable`.

- Make sure that the inputs included in the original transaction have never been seen before.
  - This prevent probing attacks.
  - This prevent reentrant payjoin, where a sender attempts to use payjoin transaction as a new original transaction for a new payjoin.

*: Interactive receivers are not required to validate the original PSBT because they are not exposed to probing attacks.

Sender's payjoin proposal checklist

The sender should check the payjoin proposal before signing it to prevent a malicious receiver from stealing money.

- Verify that the absolute fee of the payjoin proposal is equals or higher than the original PSBT.
- If the receiver's BIP21 signalled `pjos=0`, disable payment output substitution.
- Verify that the transaction version, and the nLockTime are unchanged.
- Check that the sender's inputs' sequence numbers are unchanged.
- For each inputs in the proposal:
  - Verify that no keypaths is in the PSBT input
  - Verify that no partial signature has been filled
  - If it is one of the sender's input
    - Verify that input's sequence is unchanged.
    - Verify the PSBT input is not finalized
    - Verify that `non_witness_utxo` and `witness_utxo` are not specified.
  - If it is one of the receiver's input
    - Verify the PSBT input is finalized
    - Verify that `non_witness_utxo` or `witness_utxo` are filled in.
  - Verify that the payjoin proposal did not introduced mixed input's sequence.
  - Verify that the payjoin proposal did not introduced mixed input's type.
  - Verify that all of sender's inputs from the original PSBT are in the proposal.
- For each outputs in the proposal:
  - Verify that no keypaths is in the PSBT output
  - If the output is the fee output:
    - The amount that was substracted from the output's value is less than or equal to `maxadditionalfeecontribution`. Let's call this amount `actual contribution`.
    - Make sure the actual contribution is only paying fee: The `actual contribution` is less than or equals to the difference of absolute fee between the payjoin proposal and the original PSBT.
    - Make sure the actual contribution is only paying for fee incurred

by additional inputs: `actual contribution` is less than or equals to `originalPSBTFeeRate * vsize(sender_input_type) * (count(payjoin_proposal_inputs) - count(original_psbt_inputs))`. (see Fee output section)
  - If the output is the payment output and payment output substitution is allowed.
    - Do not make any check
  - Else
    - Make sure the output's value did not decrease.
  - Verify that all sender's outputs (ie, all outputs except the output actually paid to the receiver) from the original PSBT are in the proposal.
- Once the proposal is signed, if `minfeerate` was specified, check that the fee rate of the payjoin transaction is not less than this value.

The sender must be careful to only sign the inputs that were present in the original PSBT and nothing else.

Note:

- The sender must allow the receiver to add/remove or modify the receiver's own outputs. (if payment output substitution is disabled, the receiver's outputs must not be removed or decreased in value)
- The sender should allow the receiver to not add any inputs. This is useful for the receiver to change the paymout output scriptPubKey type.
- If no input have been added, the sender's wallet implementation should accept the payjoin proposal, but not mark the transaction as an actual payjoin in the user interface.

Our method of checking the fee allows the receiver and the sender to batch payments in the payjoin transaction. It also allows the receiver to pay the fee for batching adding his own outputs.

Rationale

There is several consequences of our proposal:

- The receiver can bump the fee of the original transaction.
- The receiver can modify the outputs of the original PSBT.
- The sender must provide the UTXO information (Witness or previous transaction) in the PSBT.

Respecting the minimum relay fee policy

To be properly relayed, a Bitcoin transaction needs to pay at least 1 satoshi per virtual byte. When blocks are not full, the original transaction might already at the minimum relay fee rate (currently 1 satoshi per virtual byte), so if the receiver adds their own input, they need to make sure the fee is increased such that the rate

does not drop below the minimum relay fee rate. In such case, the sender must set both `maxadditionalfeecontribution=` and `additionalfeeoutputindex=`.

See the Fee output section for more information.

We also recommend the sender to set `minfeerate=`, as the sender's node policy might be different from the receiver's policy.

Defeating heuristics based on the fee calculation

Most wallets are creating a round fee rate (like 2 sat/b). If the payjoin transaction's fee was not increased by the added size, then those payjoin transactions could easily be identifiable on the blockchain.

Not only would those transactions stand out by not having a round fee (like 1.87 sat/b), but any suspicion of payjoin could be confirmed by checking if removing one input would create a round fee rate. In such case, the sender must set both `maxadditionalfeecontribution=` and `additionalfeeoutputindex=`.

The recommended value `maxadditionalfeecontribution=` is explained in the Fee output section. We also recommend the sender to set `minfeerate=`, as the sender's node policy might be different from the receiver's policy.

Receiver does not need to be a full node

Because the receiver needs to bump the fee to keep the same fee rate as the original PSBT, it needs the input's UTXO information to know what is the original fee rate. Without PSBT, light wallets like Wasabi Wallet would not be able to receive a payjoin transaction.

The validation (policy and consensus) of the original transaction is optional: a receiver without a full node can decide to create the payjoin transaction and automatically broadcast the original transaction after a timeout of 1 minute, and only verify that it has been propagated in the network.

However, non-interactive receivers (like a payment processor) need to verify the transaction to prevent UTXO probing attacks.

This is not a concern for interactive receivers like Wasabi Wallet, because those receivers can just limit the number of original PSBT proposals of a specific address to one. With such wallets, the attacker has no way to generate new deposit addresses to probe the UTXOs.

Spare change donation

Small change inside wallets are detrimental to privacy. Mixers like Wasabi wallet, because of its protocol, eventually generate such small change.

A common way to protect your privacy is to donate those spare changes, to deposit them in an exchange or on your favorite merchant's store account. Those

kind of transactions can easily be spotted on the blockchain: There is only one output.

However, if you donate via payjoin, it will look like a normal transaction.

On top of this the receiver can poison analysis by randomly faking a round amount of satoshi for the additional output.

Payment output substitution

Unless disallowed by sender explicitely via ‘disableoutputsubstitution=true‘ or by the BIP21 url via query parameter the ‘pjos=0‘, the receiver is free to decrease the amount, remove, or change the scriptPubKey output paying to himself. Note that if payment output substitution is disallowed, the reveiver can still increase the amount of the output. (See the reference implementation)

For example, if the sender's scriptPubKey type is P2WPKH while the receiver's payment output in the original PSBT is P2SH, then the receiver can substitute the payment output to be P2WPKH to match the sender's scriptPubKey type.

Unsecured payjoin server

A receiver might run the payment server (generating the BIP21 invoice) on a different server than the payjoin server, which could be less trusted than the payment server.

In such case, the payment server can signal to the sender, via the BIP21 parameter `pjos=0`, that they MUST disallow payment output substitution. A compromised payjoin server could steal the hot wallet outputs of the receiver, but would not be able to re-route payment to himself.

Impacted heuristics

Our proposal of payjoin is breaking the following blockchain heuristics:

- Common inputs heuristics.

Because payjoin is mixing the inputs of the sender and receiver, this heuristic becomes unreliable.

- Change identification from scriptPubKey type heuristics

When Alice pays Bob, if Alice is using P2SH but Bob's deposit address is P2WPKH, the heuristic would assume that the P2SH output is the change address of Alice. This is now however a broken assumption, as the payjoin receiver has the freedom to mislead analytics by purposefully changing the invoice's address in the payjoin transaction.

See payment output substitution.

- Change identification from round change amount

If Alice pays Bob, she might be tempted to pay him a round amount, like `1.23000000 BTC`. When this happens, blockchain analysis often identifies the output without the round amount as the change of the transaction.

For this reason, during a spare change case, the receiver may add an output with a rounded amount randomly.

Attack vectors

On the receiver side: UTXO probing attack

When the receiver creates a payjoin proposal, they expose one or more inputs belonging to them.

An attacker could create multiple original transactions in order to learn the UTXOs of the receiver, while not broadcasting the payjoin proposal.

While we cannot prevent this type of attack entirely, we implemented the following mitigations:

- When the receiver detects an original transaction being broadcast, or if the receiver detects that the original transaction has been double spent, then they will reuse the UTXO that was exposed for the next payjoin.
- While the exposed UTXO will be reused in priority to not leak other UTXOs, there is no strong guarantee about it. This prevents the attacker from detecting with certainty the next payjoin of the merchant to another peer.

Note that probing attacks are only a problem for automated payment systems such as BTCPay Server. End-user wallets with payjoin capabilities are not affected, as the attacker can't create multiple invoices to force the receiver to expose their UTXOs.

On the sender side: Double payment risk for hardware wallets

For a successful payjoin to happen, the sender needs to sign two transactions double spending each other: The original transaction and the payjoin proposal.

The sender's software wallet can verify that the payjoin proposal is legitimate by the sender's checklist.

However, a hardware wallet can't verify that this is indeed the case. This means that the security guarantee of the hardware wallet is decreased. If the sender's software is compromised, the hardware wallet would sign two valid transactions, thus sending two payments.

Without payjoin, the maximum amount of money that could be lost by a compromised software is equal to one payment (via payment output substitution). Note that the sender can disallow payment output substitution by using the optional parameter `disableoutputsubstitution=true`.

With payjoin, the maximum amount of money that can be lost is equal to two payments.

Reference sender's implementation

Here is pseudo code of a sender implementation. `RequestPayjoin` takes the bip21 URI of the payment, the wallet and the `signedPSBT`.

The `signedPSBT` represents a PSBT which has been fully signed, but not yet finalized. We then prepare `originalPSBT` from the `signedPSBT` via the `CreateOriginalPSBT` function and get back the `proposal`.

While we verify the `proposal`, we also import into it informations about our own inputs and outputs from the `signedPSBT`. At the end of this `RequestPayjoin`, the proposal is verified and ready to be signed.

We logged the different PSBT involved, and show the result in our test vectors.

```
public async Task<PSBT> RequestPayjoin(
    BIP21Uri bip21,
    Wallet wallet,
    PSBT signedPSBT,
    PayjoinClientParameters optionalParameters)
{
    Log("Unfinalized signed PSBT" + signedPSBT);
    // Extracting the pj link.
    var endpoint = bip21.ExtractPayjointEndpoint();
    if (signedPSBT.IsAllFinalized())
        throw new InvalidOperationException("The original PSBT should not be finalized.");
    ScriptPubKeyType inputScriptType = wallet.ScriptPubKeyType();
    PSBTOutput feePSBTOutput = null;

    bool allowOutputSubstitution = !optionalParameters.DisableOutputSubstitution;
    if (bip21.Parameters.Contains("pjos") && bip21.Parameters["pjos"] == "0")
        allowOutputSubstitution = false;

    if (optionalParameters.AdditionalFeeOutputIndex != null && optionalParameters.MaxAdditio
        feePSBTOutput = signedPSBT.Outputs[optionalParameters.AdditionalFeeOutputIndex];
    Script paymentScriptPubKey = bip21.Address == null ? null : bip21.Address.ScriptPubKey;
    decimal originalFee = signedPSBT.GetFee();
    PSBT originalPSBT = CreateOriginalPSBT(signedPSBT);
    Transaction originalGlobalTx = signedPSBT.GetGlobalTransaction();
    TxOut feeOutput = feePSBTOutput == null ? null : originalGlobalTx.Outputs[feePSBTOutput.
    var originalInputs = new Queue<(TxIn OriginalTxIn, PSBTInput SignedPSBTInput)>();
    for (int i = 0; i < originalGlobalTx.Inputs.Count; i++)
    {
        originalInputs.Enqueue((originalGlobalTx.Inputs[i], signedPSBT.Inputs[i]));
    }
```

```
var originalOutputs = new Queue<(TxOut OriginalTxOut, PSBTOutput SignedPSBTOutput)>();
for (int i = 0; i < originalGlobalTx.Outputs.Count; i++)
{
  originalOutputs.Enqueue((originalGlobalTx.Outputs[i], signedPSBT.Outputs[i]));
}
// Add the client side query string parameters
endpoint = ApplyOptionalParameters(endpoint, optionalParameters);
Log("original PSBT" + originalPSBT);
PSBT proposal = await SendOriginalTransaction(endpoint, originalPSBT, cancellationToken)
Log("payjoin proposal" + proposal);
// Checking that the PSBT of the receiver is clean
if (proposal.GlobalXPubs.Any())
{
    throw new PayjoinSenderException("GlobalXPubs should not be included in the receiver
}
////////////

if (proposal.CheckSanity() is List<PSBTError> errors && errors.Count > 0)
    throw new PayjoinSenderException($"The proposal PSBT is not sane ({errors[0]})");

var proposalGlobalTx = proposal.GetGlobalTransaction();
// Verify that the transaction version, and nLockTime are unchanged.
if (proposalGlobalTx.Version != originalGlobalTx.Version)
    throw new PayjoinSenderException($"The proposal PSBT changed the transaction version
if (proposalGlobalTx.LockTime != originalGlobalTx.LockTime)
    throw new PayjoinSenderException($"The proposal PSBT changed the nLocktime");

HashSet<Sequence> sequences = new HashSet<Sequence>();
// For each inputs in the proposal:
foreach (PSBTInput proposedPSBTInput in proposal.Inputs)
{
    if (proposedPSBTInput.HDKeyPaths.Count != 0)
        throw new PayjoinSenderException("The receiver added keypaths to an input");
    if (proposedPSBTInput.PartialSigs.Count != 0)
        throw new PayjoinSenderException("The receiver added partial signatures to an in
    PSBTInput proposedTxIn = proposalGlobalTx.Inputs.FindIndexedInput(proposedPSBTInput.
    bool isOurInput = originalInputs.Count > 0 && originalInputs.Peek().OriginalTxIn.Pre
    // If it is one of our input
    if (isOurInput)
    {
        OutPoint inputPrevout = ourPrevouts.Dequeue();
        TxIn originalTxin = originalGlobalTx.Inputs.FromOutpoint(inputPrevout);
        PSBTInput originalPSBTInput = originalPSBT.Inputs.FromOutpoint(inputPrevout);
        // Verify that sequence is unchanged.
        if (input.OriginalTxIn.Sequence != proposedTxIn.Sequence)
            throw new PayjoinSenderException("The proposedTxIn modified the sequence of
```

```
            // Verify the PSBT input is not finalized
            if (proposedPSBTInput.IsFinalized())
                throw new PayjoinSenderException("The receiver finalized one of our inputs")
            // Verify that <code>non_witness_utxo</code> and <code>witness_utxo</code> are n
            if (proposedPSBTInput.NonWitnessUtxo != null || proposedPSBTInput.WitnessUtxo !=
                throw new PayjoinSenderException("The receiver added non_witness_utxo or wit
            sequences.Add(proposedTxIn.Sequence);

            // Fill up the info from the original PSBT input so we can sign and get fees.
            proposedPSBTInput.NonWitnessUtxo = input.SignedPSBTInput.NonWitnessUtxo;
            proposedPSBTInput.WitnessUtxo = input.SignedPSBTInput.WitnessUtxo;
            // We fill up information we had on the signed PSBT, so we can sign it.
            foreach (var hdKey in input.SignedPSBTInput.HDKeyPaths)
            proposedPSBTInput.HDKeyPaths.Add(hdKey.Key, hdKey.Value);
            proposedPSBTInput.RedeemScript = signedPSBTInput.RedeemScript;
            proposedPSBTInput.RedeemScript = input.SignedPSBTInput.RedeemScript;
        }
        else
        {
            // Verify the PSBT input is finalized
            if (!proposedPSBTInput.IsFinalized())
                throw new PayjoinSenderException("The receiver did not finalized one of thei
            // Verify that non_witness_utxo or witness_utxo are filled in.
            if (proposedPSBTInput.NonWitnessUtxo == null && proposedPSBTInput.WitnessUtxo ==
                throw new PayjoinSenderException("The receiver did not specify non_witness_u
            sequences.Add(proposedTxIn.Sequence);
            // Verify that the payjoin proposal did not introduced mixed inputs' type.
            if (inputScriptType != proposedPSBTInput.GetInputScriptPubKeyType())
                throw new PayjoinSenderException("Mixed input type detected in the proposal'
        }
    }

    // Verify that all of sender's inputs from the original PSBT are in the proposal.
    if (originalInputs.Count != 0)
        throw new PayjoinSenderException("Some of our inputs are not included in the proposa

    // Verify that the payjoin proposal did not introduced mixed inputs' sequence.
    if (sequences.Count != 1)
        throw new PayjoinSenderException("Mixed sequence detected in the proposal");

    decimal newFee = proposal.GetFee();
    decimal additionalFee = newFee - originalFee;
    if (additionalFee < 0)
      throw new PayjoinSenderException("The receiver decreased absolute fee");
    // For each outputs in the proposal:
    foreach (PSBTOutput proposedPSBTOutput in proposal.Outputs)
```

```
{
    // Verify that no keypaths is in the PSBT output
    if (proposedPSBTOutput.HDKeyPaths.Count != 0)
        throw new PayjoinSenderException("The receiver added keypaths to an output");
    if (originalOutputs.Count == 0)
            continue;
    var originalOutput = originalOutputs.Peek();
    bool isOriginalOutput = originalOutput.OriginalTxOut.ScriptPubKey == proposedPSBTOut
    bool substitutedOutput = !isOriginalOutput &&
                             allowOutputSubstitution &&
                             originalOutput.OriginalTxOut.ScriptPubKey == paymentScriptPu
    if (isOriginalOutput || substitutedOutput)
    {
        originalOutputs.Dequeue();
        if (output.OriginalTxOut == feeOutput)
        {
            var actualContribution = feeOutput.Value - proposedPSBTOutput.Value;
            // The amount that was substracted from the output's value is less than or e
            if (actualContribution > optionalParameters.MaxAdditionalFeeContribution)
                throw new PayjoinSenderException("The actual contribution is more than m
            // Make sure the actual contribution is only paying fee
            if (actualContribution > additionalFee)
                throw new PayjoinSenderException("The actual contribution is not only pa
            // Make sure the actual contribution is only paying for fee incurred by addi
            int additionalInputsCount = proposalGlobalTx.Inputs.Count - originalGlobalTx
            if (actualContribution > originalFeeRate * GetVirtualSize(inputScriptType) *
                throw new PayjoinSenderException("The actual contribution is not only pa
        }
        else if (allowOutputSubstitution && output.OriginalTxOut.ScriptPubKey == payment
        {
            // That's the payment output, the receiver may have changed it.
        }
        else
        {
            if (originalOutput.OriginalTxOut.Value > proposedPSBTOutput.Value)
                throw new PayjoinSenderException("The receiver decreased the value of or
        }
        // We fill up information we had on the signed PSBT, so we can sign it.
        foreach (var hdKey in output.SignedPSBTOutput.HDKeyPaths)
            proposedPSBTOutput.HDKeyPaths.Add(hdKey.Key, hdKey.Value);
        proposedPSBTOutput.RedeemScript = output.SignedPSBTOutput.RedeemScript;
    }
}
// Verify that all of sender's outputs from the original PSBT are in the proposal.
if (originalOutputs.Count != 0)
{
```

```
        // The payment output may have been substituted
        if (!allowOutputSubstitution ||
            originalOutputs.Count != 1 ||
            originalOutputs.Dequeue().OriginalTxOut.ScriptPubKey != paymentScriptPubKey)
            {
                throw new PayjoinSenderException("Some of our outputs are not included in th
            }
    }

    // After signing this proposal, we should check if minfeerate is respected.
    Log("payjoin proposal filled with sender's information" + proposal);
    return proposal;
}

int GetVirtualSize(ScriptPubKeyType? scriptPubKeyType)
{
    switch (scriptPubKeyType)
    {
        case ScriptPubKeyType.Legacy:
            return 148;
        case ScriptPubKeyType.Segwit:
            return 68;
        case ScriptPubKeyType.SegwitP2SH:
            return 91;
        default:
            return 110;
    }
}

// Finalize the signedPSBT and remove confidential information
PSBT CreateOriginalPSBT(PSBT signedPSBT)
{
    var original = signedPSBT.Clone();
    original = original.Finalize();
    foreach (var input in original.Inputs)
    {
        input.HDKeyPaths.Clear();
        input.PartialSigs.Clear();
        input.Unknown.Clear();
    }
    foreach (var output in original.Outputs)
    {
        output.Unknown.Clear();
        output.HDKeyPaths.Clear();
    }
    original.GlobalXPubs.Clear();
```

```
    return original;
}
```

Test vectors

A successful exchange with:

InputScriptType	Orginal PSBT Fee rate	maxadditionalfeecontribution	additionalfeeoutputindex
P2SH-P2WPKH	2 sat/vbyte	0.00000182	0

```
Unfinalized signed PSBT
cHNidP8BAHMCAAAAAY8nutGgJdyYGXWiBEb45Hoe9lWGbkxh/6bNiOJdCDuDAAAAAAD+////AtyVuAUAAAAAF6kUHehJ
Original PSBT
cHNidP8BAHMCAAAAAY8nutGgJdyYGXWiBEb45Hoe9lWGbkxh/6bNiOJdCDuDAAAAAAD+////AtyVuAUAAAAAF6kUHehJ
payjoin proposal
cHNidP8BAJwCAAAAAo8nutGgJdyYGXWiBEb45Hoe9lWGbkxh/6bNiOJdCDuDAAAAAAD+////jye60aAl3JgZdaIERvjk
payjoin proposal filled with sender's information
cHNidP8BAJwCAAAAAo8nutGgJdyYGXWiBEb45Hoe9lWGbkxh/6bNiOJdCDuDAAAAAAD+////jye60aAl3JgZdaIERvjk
```

Implementations

- BlueWallet is in the process of implementing the protocol.
- BTCPay Server has implemented sender and receiver side of this protocol.
- Wasabi Wallet has merged sender's support.
- Join Market has implemented sender and receiver side of this protocol.
- JavaScript sender implementation.

Backward compatibility

The receivers are advertising payjoin capabilities through BIP21's URI Scheme.

Senders not supporting payjoin will just ignore the `pj` variable and thus, will proceed to normal payment.

Special thanks

Special thanks to Kukks for developing the initial support to BTCPay Server, to junderw, AdamISZ, lukechilds, ncoelho, nopara73, lontivero, yahiheb, SomberNight, andrewkozlik, instagibbs, RHavar for all the feedback we received since our first implementation. Thanks again to RHavar who wrote the BIP79 Bustapay proposal, this gave a good starting point for our proposal.