```
BIP: 351
Layer: Applications
Title: Private Payments
Author: Alfred Hodler <alfred_hodler@protonmail.com>
        Clark Moody <clark@clarkmoody.com>
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0351
Status: Draft
Type: Informational
Created: 2022-07-10
License: MIT
```

Abstract

This BIP makes it possible for two parties to transact using addresses that only they can calculate. This is done using exclusively on-chain methods and in a manner that minimizes blockchain footprint. Receiving parties can share their payment codes publicly without a loss of privacy, as every sender will calculate a unique set of addresses for each payment code.

Motivation

A recipient that wishes to receive funds privately has several options. Each has tradeoffs in terms of chain analysis potential, recoverability, and wallet complexity.

Sharing a static address works well enough for one-time payments between two parties as long as the address is shared through a private channel. It does not work well for recurring payments because address reuse leads to a loss of privacy. Using this method for donations exacerbates the problem since the address will serve as a focal point for data collection and analysis. Wallets must not reissue the same address to multiple recipients.

Sharing a BIP32 extended public key works for recurring payments between two parties only. The same key cannot be shared to any other party without leaking the chain of payments. Furthermore, an extended public key does not say anything about address types and makes it possible for a sender to send to a script that a recipient cannot spend from. Alternate version bytes have been proposed to specify address types, but wallet adoption is limited.

Sharing a BIP380 descriptor containing an extended public key solves the address type issue from sharing a raw BIP32 extended key. The drawback is that descriptor support is not widespread, especially in mobile wallets.

Using a payment server works in the case of recipients that have the resources to set up and maintain a payment server that will generate a fresh address for each payment. These are usually businesses and the method is usually out of reach for the average user. The centralized server is vulnerable to takedown remotely and physically.

Sharing a BIP47 payment code addresses most of the above shortcomings. However, it introduces the following problems:

- The BIP uses a notification mechanism that relies on publicly known per-recipient notification addresses. If Alice wants to send funds to Bob, she has to use the same notification address that everyone else uses to notify Bob. If Alice is not careful with coin selection, i.e. ensuring that her notification UTXO is not linked to her, she will publicly expose herself as someone who is trying to send funds to Bob and their relationship becomes permanently visible on the blockchain.
- The BIP does not say anything about address types. Receiving wallets therefore have to watch all address types that can be created from a single public key. Even then, a sender could send to a script that a receipient cannot spend from.

Method

When Alice wants to start paying Bob in private, she imports his payment code into a compatible wallet. Her wallet extracts Bob's public key from the payment code and sends a notification transaction. If Bob finds a notification transaction addressed to himself, he imports Alice's public key contained therein and stores it. Bob then performs ECDH using Alice's public key and his own private key in order to calculate a common set of addresses to watch. Alice calculates the same set of addresses on her end and uses them to send coins to Bob. If Alice engages in coin control, both the initial notification transaction and subsequent payment transactions cannot be attributed to either party. Even if Alice uses coins that are already associated with her, chain analysis will identify her as a sender but Bob's privacy will remain entirely preserved.

Specification

Definitions

- Alice: sender
- Bob: recipient
- Payment code: static string that Bob generates and shares with others so that he can receive payments
- P: public key contained in Bob's payment code
- p: private key associated with Bob's public key P
- N: extended public key used by Alice to derive child keys for each Bob she wants to transact with
- n: private key associated with Alice's public key N
- x: Alice's secret recipient index, unique for each Bob
- N_x: child public key derived from N at index x (non-hardened)
- n_x: private key associated with N_x
- c: Alice's transaction count toward Bob

- P_c: Bob's public key at index c
- p_c: Bob's private key at index c
- A_c: Bob's receive address at index c
- H: SHA256 hash function
- *: EC multiplication
- +: EC addition
- /: string concatenation
- *[a..b]*: string slicing (inclusive of a, exclusive of b)

Public Key Derivation Path

The derivation path for this BIP follows BIP44. The following BIP32 path levels are defined:

```
m / purpose' / coin_type' / account'
```

`purpose` is set to 351.

(p, P) and *(n, N)* are keys associated with the above path, depending on which side is performing the calculation.

N_x keys are the direct non-hardened children of *N*. For instance, the path of N_0 from *N* is *m / 0*.

Payment Code Structure and Encoding

- bytes `[0..2]`: address type flags (2 bytes)
- bytes `[2..35]`: compressed public key P (33 bytes)

Payment codes are encoded in bech32m and the human readable part is "pay" for mainnet and "payt" for testnet (all types), resulting in payment codes that look like "pay1cqqq8d29g0a7m8ghmycqk5yv24mfh3xg8ptzqcn8xz6d2tjl8ccdnfkpjl7p84".

Address Types

Address type flags determine which address types a payment code accepts. This is represented by big-endian ordered 16 bits. For instance, a hypothetical payment code that handles all address types will have all defined bits set to 1 (`0xffff`).

Currently defined flags:

Address Type	Flag	Flag Value	Ordinal Value
P2PKH	`1 << 0`	0x0001	0
P2WPKH	`1 << 1`	0x0002	1
P2TR	`1 << 2`	0x0004	2

The remaining flags are reserved for future address types.

While payment codes use 2-byte bitflag arrays, notifications use ordinal values in the form of a single byte.

All keys are compressed. Using uncompressed keys at any point is illegal.

Notifications

Notifications are performed by publishing transactions that contain a 40-byte `OP_RETURN` output. The value of the `OP_RETURN` is constructed using the following formula:

search_key | notification_code | N_x *| address_type*

- *search_key* equals "PP" and is a static ASCII-encoded string (2 bytes)
- *notification_code* is $H(n_x * P)[0..4]$ (4 bytes)
- N_x is the unique public key a sender is using for a particular recipient (33 bytes)
- *address_type* is the **ordinal** value of a single address type that a sender wants to send to (1 byte). This must be selected from the recepient's accepted address types.

When Alice wants to notify Bob that he will receive future payments from her, she performs the following procedure:

1. Assigns an unused, unique index x to Bob (*0* if Bob is the first party she is notifying).
2. Calculates a 4-byte notification code: *notification_code* $= H(n_x * P)[0..4]$
3. Commits to one of Bob's accepted address types by choosing its ordinal value. Going forward Alice must not send to address types other than the one she committed to in the notification.
4. Constructs a notification payload by concatenating the above values according to the formula.
5. Selects any UTXO in her wallet, preferably not associated with her.
6. Sends a transaction including an `OP_RETURN` output whose value is set to the constructed payload.

When Bob notices a 40-byte `OP_RETURN` starting with *search key*, he performs the following procedure:

1. Breaks down the payload into its four constituent parts.
2. Discards the *search_key* (item #0).
3. Selects N_x (item #2) and performs $H(N_x * p)$ (Bob does not know the value of x). Bob takes the first four bytes of the calculated value.
4. If the four bytes match the notification value (item #1), Bob found a notification addressed to himself and stores N_x together with *address_type*.
5. If this process fails for any reason, Bob assumes a spurious notification or one not addressed to himself and gives up.

Since changing x yields a completely different sender identity, Alice can always re-notify Bob from a different index when she does not want to be associated with her previous identity. Alice can also re-notify Bob when she wants to start sending to a different address type. Bob must be able to update his watchlist in that case and he can stop watching addresses associated with the old address type.

Out-of-band notifications between Alice and Bob are legal (in fact, they may not be prevented), but in that case Bob loses the ability to restore his wallet from `OP_RETURN` outputs embedded in the blockchain. In that case, Bob has the burden of keeping a valid backup of any out-of-band notifications.

Allowing Notification Collisions

Since *notification_code* is a 4-byte truncation of the full value, Bob has a 1 in ~4.3 billion chance of detecting a spurious notification. This is considered acceptable because the cost of doing so is adding a few more addresses to Bob's watchlist. The benefit of this approach is that is saves 28 bytes per notification.

Scanning Requirement

There is a scanning requirement on the recipient side in that the recipient must have access to full blocks in order to be able to search them for OP_RETURN outputs containing notifications. For more information on how light clients can get around this limitation and still use the standard, see Appendix B.

Recipients that do not want to decode raw block data can quickly search for notifications in a block by looking for the following byte array: `[106, 40, 80, 80]`. The first two bytes represent *OP_RETURN* and *OP_PUSHBYTES_40*, followed by the ASCII value of *search_key*.

Transacting

Alice initializes counter c which is unique to Bob and increments with each transaction. c is a 64-bit integer and must be inputted into a hasher as a big-endian encoded array of 8 bytes.

1. Alice calculates a secret point (constant between Alice and Bob):

$S = n_x * P$

2. Alice calculates a shared secret:

$s = H(S \mid c)$

3. Alice calculates Bob's ephemeral public key and its associated address where the funds will be sent:

$P_c = P + s*G$

4. Alice constructs an address using the key P_c, using one of the address types she committed to in the notification transaction.

Bob constructs his watchlist by mirroring this process on his end, except that his method of calculating S is:

$S = N_x * p$

When Bob wants to spend from such addresses, he calculates his private keys in the following manner:

$p_c = p + s$

Backward Compatibility

Private Payments is a new standard which is not compatible with any previous standard based on static payment codes, such as BIP47.

While the standard does not support versioning, it reserves unused bits in the address type bitflag array which can be allocated to new address types once they are deemed ubiquitous. Older payment codes (i.e. those generated when fewer address types were available) are readable by software supporting new address types. The reverse is also supported since older software will ignore newer address type flags that are not understood.

Appendix A: Test Vectors

Alice's Wallet

BIP32 seed: 0xfe

Master xprv: xprv9s21ZrQH143K2qVytoy3eZSSuc1gfzFrkV4bgoHzYTkgge4UoNP62eV8jkHYNqddaaefpnjwkz

n: xprv9zNFGn56Wm1s89ycTCg4hB615ehu6ZvNL4mxUEAL28pNhBAb6SZgLdsgmQd1ECgAiCjy6XxTTRyBd

N: xpub6DMbgHbzM8aALe45ZED54K2jdgYPW2eDhHhZGcZwaUMMZyVjdysvtSCAcfPYiqB5Zw41EyLWPxC

x: 0

$\mathbf{n_x}$: be9518016ec15762877de7d2ce7367a2087cf5682e72bbffa89535d73bb42f40

$\mathbf{N_x}$: 02e3217349724307eed5514b53b1f53f0802672a9913d9bbb76afecc86be23f464

Bob's Wallet

BIP32 seed: 0xff

Master xprv: xprv9s21ZrQH143K47bRNtc26e8Gb3wkUiJ4fH3ewYgJeiGABp7vQtTKsLBzHM2fsfiK7Er6uMr

p: 0x26c610e7d0ed4395be3f0664073d66b0a3442b49e1ec13faf2dd9b7d3c335441

P: 0x0302be8bff520f35fae3439f245c52afb9085a7bf62d099c1f5e9e1b15a7e2121a

Accepted scripts: 0x03 (legacy + segwit) (0x01 | 0x02)

Payment code: pay1qqpsxq4730l4yre4lt3588eyt3f2lwggtfalvtgfns04a8smzkn7yys6xv2gs8

Alice notifying Bob

S: 0x02c0892d6ba30b5b1eafebd47172e46d358721f294698f9f59b4d96b781da09a62

Notification code: 0x49cb55bb

Address type commitment: 1 (segwit)

Notification output script: OP_RETURN OP_PUSHBYTES_40 505049cb55bb02e3217349724307eed5514b53b1f53f0802672a9913d9bbb76afecc86be23f46401

Alice sending to Bob

c: 0

s: 0x5dbe5efee4a5b9df73708241858f2bf7ec65f141dbd229ea8e2f9f51804a18f2

s*G: 0x039362033c1bc3f05e081d4d7f76d5ffebde349b0f6a4d2e8ffc5c065c17233247

P_c: 0x03e669bd1705691a080840b07d76713d040934a37f2e8dde2fe02f5d3286a49219

A_c: bc1qw7ld5h9tj2ruwxqvetznjfq9g5jyp0gjhrs30w

Bob spending

c: 0

p_c: 0x84846fe6b592fd7531af88a58ccc92a88faa1c8bbdbe3de5810d3acebc7d6d33

Appendix B: Potential OP_RETURN Services

Compact Block Filters, as formulated in BIP-0158, do not cover `OP_RETURN` data payloads. In support of light wallets, an external service could publish transaction proofs for all transactions that include the tagged notification payload. Light wallets would download all such transactions, filter for matches against their payment code, then verify the transaction proofs against the block headers obtained over the P2P network.

Appendix C: Potential Notification Transaction Services

No specific instruction is given as to the details of the notification transaction beyond simply including the single `OP_RETURN` payload. Since no restriction exists for other inputs or outputs of this transaction, there is an opportunity for an external service to include this payload in a transaction completely unrelated to Alice's wallet. Such a service could charge a fee out-of-band to help cover fees.

Another opportunity exists for an existing business to attach notification payloads to transactions sent during the normal course of operations. Large withdrawal transactions from mining pools or exchanges could include a marginal notification payload without affecting overall fees.

Reference Implementation

Reference implementation is available at https://github.com/private-payments/rust-private-payments

Reference

- BIP32 - Hierarchical Deterministic Wallets
- BIP43 - Purpose Field for Deterministic Wallets
- BIP44 - Multi-Account Hierarchy for Deterministic Wallets
- BIP47 - Reusable Payment Codes for Hierarchical Deterministic Wallets
- BIP157 - Client Side Block Filtering
- BIP158 - Compact Block Filters for Light Clients
- BIP47 Prague Discussion (acknowledgements: @rubensomsen, @afilini, @kixunil)