easypaysy

Improving Bitcoin's UX by replacing addresses with account IDs

José Femenías Cañuelo
jose.femenias@gmail.com
December, 1st, 2019

Abstract

Even after the advent of the more user-friendly bech32 [1] format, one of the major pain points in the UX of Bitcoin [2] is the need to exchange and use a different address for each payment. We present *easypaysy*, a layer-two protocol for Bitcoin, that enables the creation of non-custodial accounts, directly on the blockchain. Upon opening an account, the mining process assigns it a permanent account ID -*such as btc@543847.636/577*- that becomes the destination for payments addressed to the account. Although the account ID is immutable, the *easypaysy* protocol provides mechanisms to ensure that each payment goes to a different bitcoin address, thus achieving a level of privacy similar to regular Bitcoin payments. The protocol also introduces IOC payments, that enable the sender of a payment -*instead of its recipient*- to be the one that chooses the payment address, while ensuring that only the payee has access to the corresponding private key. IOC is the foundation for non-interactive payments that offer an enhanced level of privacy. We also show how *easypaysy* accounts offer some additional benefits, such as the ability to implement pull payments, their integration with the Lightning Network, and the non-repudiable nature of their payments. Finally, we explore some scalability techniques that can give support to the creation of several billions of accounts per year.

Keywords: easypaysy, bitcoin, litecoin, layer-two, non-custodial account, ioc, repudiability, ux, pull payments, inversion of control, BIP39

1 Introduction

The use and format of bitcoin addresses has been recognized by many in the Bitcoin community as a deficient mechanism from the UX's perspective. In the rationale section of Bitcoin Improvement Proposal #13 we can read: "*...bitcoin addresses should be deprecated in favor of a more user-friendly mechanism for payments... Another criticism is that bitcoin addresses are inherently insecure because there is no identity information tied to them; if you only have a bitcoin address, how can you be certain that you're paying who or what you think you're paying?...A future BIP or BIPs should propose more user-friendly mechanisms for making payments, or for verifying that you're sending a payment to the Free Software Foundation and not Joe Random Hacker.*"

Easypaysy deals with these challenges by completely hiding bitcoin addresses from the user experience, replacing them with account IDs, that are significantly more user-friendly, permanent and secure.

2 Accounts

Accounts are a widely used metaphor for user interaction with many services, financial or otherwise: email, phone, PayPal, banking…
They provide a consistent anchor point for the owner of the account and a convenient way for others to reach the account holder through her account ID.
Although Bitcoin payments weren't originally designed with an account system in mind, its flexible architecture and permissionless nature allow for an account structure to be superimposed over the regular flow of transactions.
We propose the creation of *easypaysy*, a layer-two protocol, that brings this powerful metaphor to the blockchain, in order to solve the usability problems described in the previous section.

2.1 Definition and properties

An *easypaysy* account is a standard Bitcoin transaction, built in a specific way in accordance with the protocol, that allows blockchain payments to be directed to its fixed ID, hiding the underlying bitcoin addresses from the user experience.

Easypaysy accounts have similarities with regular bank accounts, such as:

- **ID**. Each *easypaysy* account has a unique, permanent identifier, where payments are sent to.
- **Identity**. Behind every account there is a cryptographic identity.
- **Confidentiality**[1]. Only the account holder can view its transaction history and balance.
- **Push and pull payments**[2]. In addition to traditional push payments, *easypaysy* accounts can support pull payments, such as direct debit.
- **Non-repudiation**. All *easypaysy* payments are non-repudiable.

They also have several characteristics that differentiate them from bank accounts, making them:

- **Non-custodial**. Opening, using and maintaining an *easypaysy* account is permissionless.
- **Hard to censor**. As we can see in "*2.4. Censorship*", *easypaysy* payments are practically immune to censorship.
- **Irreversible**. Payments to an *easypaysy* accounts are final, unless the output script used specifies otherwise.
- **Pseudonymous**. No real-life information is directly stored within the account.

2.2 Account attributes

In addition to its permanent ID, every *easypaysy* account contains three pieces of information:

- Identity key: A point in the Secp256k1 elliptic curve, used to sign messages and to exchange encrypted information between the payer and the payee.

- Value key: A point in the Secp256k1 elliptic curve, used as a basis for non-interactive payments (see *"3.2.1 IOC payments"*).

[1] Type_0 payments don't enjoy this property, so they are discouraged. See "*2.6.1 Payment types*".

[2] Pull payments are planned for a future release of the protocol. See "*3.1.3 Pull payments*".

- Rendezvous descriptor: A JSON document that describes the kind of payments the account holder is willing to accept and the protocols and means of contact that the payer can use to interact with the account.

The first two pieces of information -Identity key and Value key- are exposed in the signature of the transaction that -by definition- must be a 2-of-2 multisig address. The Rendezvous descriptor is serialized -in a highly compressed manner- within a mandatory OP_RETURN output.

2.3 Account ID

Easypaysy IDs are assigned quasi-randomly during the mining process, as we'll see below. Once a transaction has been incorporated into a block that has been successfully appended to the blockchain, the account will acquire an account ID, in accordance with this syntax:

btc@block_height.tx_ordinal/checksum[3]

In the expression above, **block_height** is the height of the block that contains the transaction with the account information (starting at 0, the Genesis block), **tx_ordinal** is the ordinal of the transaction within the block (starting at 0, the coinbase transaction) and checksum is a value that protects the integrity of the account ID.

There are three major ways to express an account ID: Canonical ID, Mnemonic ID and Domain ID.

2.3.1 Canonical ID

Canonical IDs use decimal numbers for their constituent parts, except for the **btc@** prefix, that is constant.

For example, an account anchored in the **636**$_{th}$ transaction of block **#543847** could have this canonical ID:

btc@543847.636/577

The checksum part is extensible, up to 4 chunks of 3 digits each, separated by hyphens, as shown in this example:

btc@543847.636/577-218-376-867

2.3.2 Mnemonic ID

Mnemonic IDs use BIP39 words as a way to express the different parts that make up the ID. The same account, shown in the previous example, can be expressed with this mnemonic ID:

btc@cancel-mind.exhibit/motion

Or, in case the account holder chooses to use the longer version of the checksum, with:

btc@cancel-mind.exhibit/motion-custom-fun-sugar

To express a number in this format, you simply convert it to base 2048, using the corresponding BIP39 word for each digit (0=abandon, 2047=zoo), concatenated with hyphens. Checksum words, however, only use even words, so, before the lookup, each value has to be multiplied by 2.

As we can see in table 1, only the words that have an even index are used as checksum words. The main purpose of this is to use a wider range of words from the dictionary. Otherwise, since checksum chunks go from 000 to 999, the last word used would be *language*, thus leaving all the words from *laptop* and onward unused.

[3] When using Testnet, the coin symbol 'btc' will be replaced with 'tbtc'.

	BIP39 word	Checksum value
0	*abandon*	000
1	~~*ability*~~	---
2	*able*	001
3	~~*about*~~	---
...		
1996	*wet*	998
1997	~~*whale*~~	---
1998	*what*	999

Table 1.- Checksum words

The odd-indexed words (*ability*, *about*,...) are not used as checksum words, so they serve perfectly as decoy words, as we'll see later in the example UI mockup (see "2.3.5 *Mixed IDs"*).

2.3.3 Domain ID

In addition to these two, wholly blockchain-centric approaches, the protocol also contemplates a third type of account ID, that relies on the Internet Domain Name System. (Please note that, while the first two forms of the account ID emerge automatically upon creating an *easypaysy* account, Domain IDs are purely optional. They are a better fit for a corporate or professional setting, due to the recurrent costs and the hassle associated with buying, configuring and maintaining a domain name in a secure way).

Domain IDs follow this format:

btc@<fully_qualified_domain_name>/checksum

2.3.3.1 Creating a Domain ID

To create a Domain ID, upon successfully activating an *easypaysy* account, users can insert the account ID with its checksum, into a TXT record of a domain of their own, so that a simple lookup will suffice for any interested party to find out the user's account.

Continuing with the last example, if the owner of that *easypaysy* account also owns the domain **example.com**, she could associate it with her *easypaysy* account, to create this Domain ID:

btc@example.com/motion-custom

It is of note that the checksum is the only part of the ID that forces an attacker that manages to hijack or spoof her DNS TXT record to redirect it to another account that has the very same checksum[4]. This becomes prohibitively expensive as the size of the checksum grows, since he needs to pay a fee for each account created, as we'll see later (see "2.3.8 *Choosing a checksum size"*).

2.3.4 Real-world identity

Easypaysy accounts don't store or link to any kind of real-world identity. The only identity behind the Identity key is that of the account ID.

Any scheme linking outwards the blockchain to a real-world identity -such as the domain name of an entity- would both increase the onchain storage needs for the account and invite all sorts of cybersquatting.

If the owner of an account wishes to publicly show her association with her *easypaysy* ID, the link is best done inward, from the real-world into the blockchain. That can be easily accomplished by using her Domain ID, as we have just seen in the previous point. Also, any real-word mechanisms commonly used for announcing bank accounts, phone

[4] See *"2.4.1 Self censorship*", for another way to fight account hijacking.

numbers, email addresses, etc., will also work with *easypaysy* IDs, so there is really no need to turn the blockchain into some sort of directory service.

2.3.5 Mixed IDs

Canonical IDs use numbers, while their Mnemonic counterpart use words. However, it is entirely possible to mix both, to offer a better user experience. For instance, when using a smaller form factor device -such as a phone- the user may be requested to first input the digits that make up the Canonical ID, then the checksum words corresponding to its Mnemonic ID. This way, the UI could show a full size numeric keypad for the first part of the data input, and a series of words for the second, as shown below.

Figures 1.a. through 1.d. present a mockup of a UI showing a possible interaction sequence, first entering the Canonical ID with a numeric keypad, then specifying the checksum by selecting them from array of labeled buttons that include the four expected checksum words (1.b. through 1.d.)

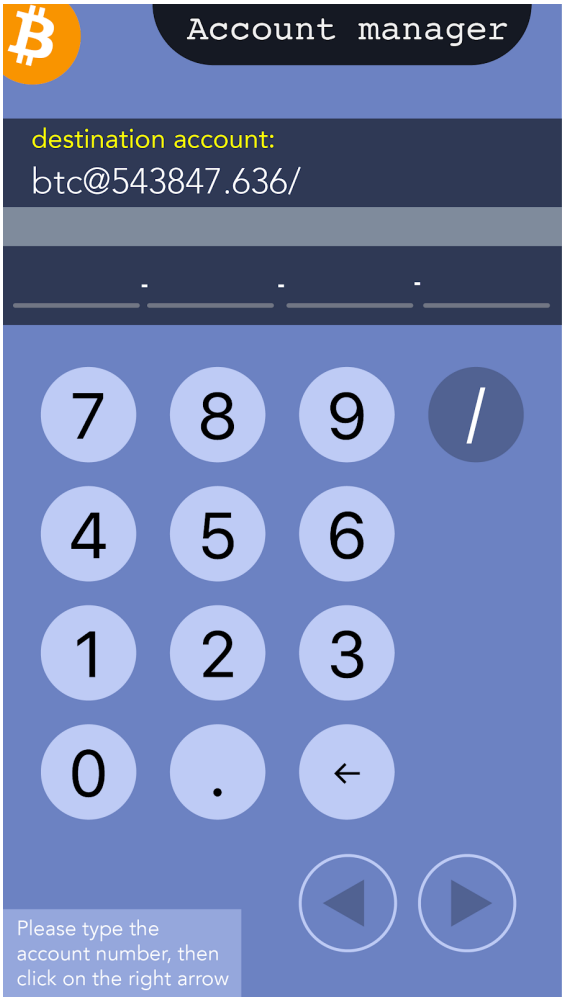

Fig. 1.a.- A numeric keyboard is used to type in the Canonical ID, minus the checksum.

The first part of the data entry is enough to detect any mistakes that point to a non-existent account. Then, the user is presented with the expected checksum words (motion, custom, fun, sugar) along with 12 decoy words, chosen at random. Just four clicks are needed to enter the checksum this way, instead of the 12 that would be required if she had to type the full set of checksum numbers using the numeric keypad.

Fig. 1.b.- A second screen presents the 4 checksum words intermixed with 12 decoy words. Please note that it also shows the Mnemonic ID, that the user didn't specify, as soon as the user clicks on the first checksum word.

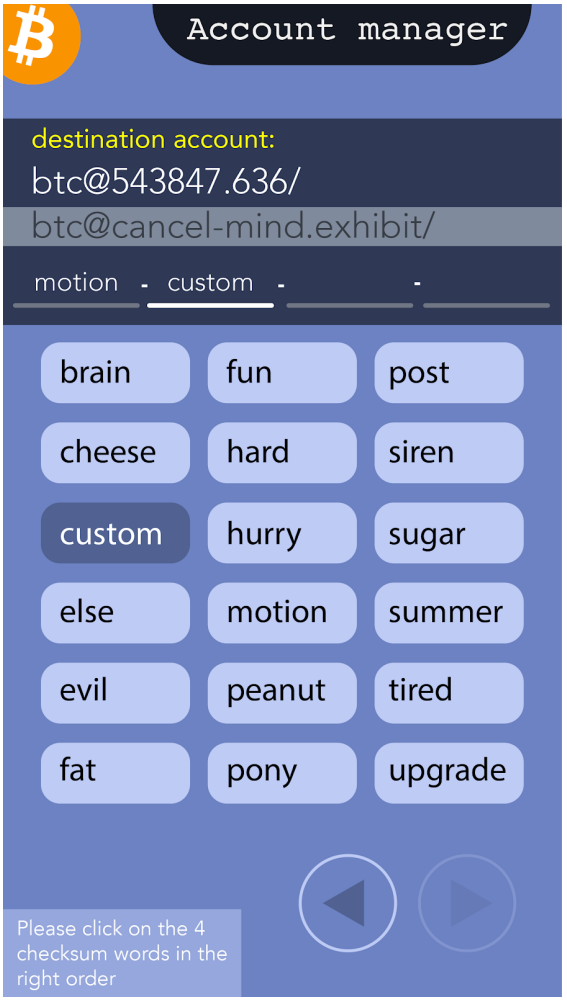

Fig. 1.c.- Clicking on the 2nd checksum word.

Fig. 1.d.- Clicking on the 3rd checksum word.

Fig. 1.e.- Clicking on the last checksum word.

2.3.6 Alias ID

The quasi-random mechanism used to assign the ID of an account, can lead to awkward combinations of words, when expressing the ID using entries from the BIP39 dictionary.

A simple way to avoid an undesired combination of words is to simply discard the account, preferably by revoking it. However, that is both wasteful and inconvenient, so it pays to offer the user other ways of dealing with this issue. The naive approach of attempting to remove from the dictionary every word with negative connotations such as *war*, *kill*, *weapon*, *stupid,* etc. quickly reveals itself as futile.

On the one hand, the number of potentially offensive words grows very soon out of hand, thus rendering the dictionary ineffective. On the other, cultural or personal preferences can instill a word or set of words with drastically different connotations. Even seemingly innocent words, such as *trophy* and *wife,* can become offensive when paired together.

Also, the very same combination of words can go from desirable to inappropriate, depending on the context. A lawyer may be happy to use the account ID **btc@divorce-wife.very**/soon while a marriage counselor would probably reject it.

Easypaysy tackles this issue by assigning every account two different Mnemonic IDs, and letting the account holder decide which one to use.

The main account ID is called *standard ID*, while the alternate receives the name *alias ID*.

Alias IDs use the same BIP39 english dictionary as standard IDs, but in reverse order (0=zoo, 2047=abandon).

Depending on the way the dictionary is sorted, the account of the previous example (**btc@1050583.1943**/829) could be identified by these two IDs:

Standard ID: **btc@divorce-wife.very**/soon
Alias ID: **btc@say-agree.artist**/cost

Upon creating a new account, the account holder, after being presented with both options, can apply her own personal preferences to choose the one that she finds more appealing.

2.3.7.1 ID disambiguation

The fact that both the standard and alias forms of an ID use the very same set of words, taken from the BIP39 dictionary, introduces a source of ambiguity. That is, given a specific ID like **btc@divorce-wife.very**/soon**,** how can a parser know for sure whether it is a standard ID, thus pointing to **btc@1050583.1943** or an alias ID, that points to **btc@3143720.104**?

The answer lies in the checksum[5]. Due to the way checksum words are selected, words used in standard ID checksums are incompatible with those of their alias ID counterpart. As a result, when the parser isolates the first checksum word -soon-, it can immediately determine that the ID is in standard form, thus removing the ambiguity.

In the mockup UI depicted in "*2.3.5 Mixed IDs*" we saw how the user types in a canonical ID that can result in two Mnemonic IDs:

Canonical:
btc@543847.636/577-218-376-867

Mnemonic (standard):
btc@cancel-mind.exhibit/motion-custom-fun-sugar

Mnemonic (alias):
btc@tell-indoor.race/hurry-siren-peanut-cheese

The two potential sets of checksum words ("motion, custom, fun, sugar" or "hurry, siren, peanut, cheese") are intermixed with eight additional words, used as decoys.

In this example, as soon as the user selected the first checksum word -motion- the app knew she was using the standard form of the Mnemonic, and it was able to display the

[5] In fact, even without resorting to the checksum, block #3143720 is much further into the future than block #1050583. So, during the about 40 years that will separate the creation of both blocks, the parser can resolve the ID unequivocally before analyzing the checksum.

proper ID, that is, **btc@cancel-mind.exhibit**, thus reassuring the user and eliminating any possible ambiguity.

2.3.7 Computing the checksum

The checksum is a critical part of an account ID, since it provides the basis to ensure that payments aren't sent to the wrong destination by mistake or malice.

Checksums are calculated by following this algorithm:

1) Calculate:

```
tx_chain = block_hash &
           merkle_root &
           wtxid
```

(where **block_hash** is the hash of the block containing the account's transaction, **merkle_root** is the root of the Merkle tree of said block, **wtxid**[6] is the double SHA256 of the serialized transaction and the **&** represents concatenation).

2) Compute:

```
tx_digest = digest(sha256(tx_chain))
```

3) Divide tx_digest in four groups of 8 bytes:

```
tx_digest_0 = tx_digest[0..7]
tx_digest_1 = tx_digest[8..15]
tx_digest_2 = tx_digest[16..23]
tx_digest_3 = tx_digest[24..31]
```

4) Finally, assign each checksum_chunk the remainder resulting from dividing each value of tx_digest_i by 1000 (**%** indicates modulus of the integer division):

```
checksum_chunk_0 = tx_digest_0 % 1000
checksum_chunk_1 = tx_digest_1 % 1000
checksum_chunk_2 = tx_digest_2 % 1000
checksum_chunk_3 = tx_digest_3 % 1000
```

[6] See https://bitcoincore.org/en/segwit_wallet_dev/

2.3.8 Choosing a checksum size

Even without the checksum, oftentimes it will be possible to detect mistakes when the user mistypes an account. Since it is to be expected that, even if *easypaysy* accounts became very popular, the majority of the transactions would not host *easypaysy* accounts, whenever a mistyped account ID points to a regular or even non-existent transaction, the software can readily identify the problem and alert the user.

However, due to the great incentive of profiting from user mistakes, it seems reasonable to think that some nefarious actors would try to game the system, for example by inserting many accounts at random, or even aiming for specific block spots, waiting to exploit typing errors from a particular account.

For example, if a popular exchange has the account **btc@80**23**00.507** it would be tempting for an adversary to create an account at **btc@80**32**00.507** or **btc@80**43**00.507**, etc.
Even though the order of a transaction within a block is out of the control of the sender of the transaction, it isn't inconceivable that a miner could agree to place a particular transaction at a requested spot, in exchange for an extra financial reward. Due to these potential threats, the checksum becomes an integral part of the account ID. Because of the way it is computed, involving the hash of the block where the account resides, it can only be known after the account's transaction has been included in a block and the block has been mined. That implies that, in practice, it is impossible to choose the checksum for a new account. Even the miners themselves can't tamper with this process, since the account's checksum is based on the block hash itself in an unpredictable manner, due to the one-way nature of the hash function.
Thus, the checksum can be seen as a random number, ranging from 000-000-000-000 to 999-999-999-999. As a result, the probability

of two different accounts having the same full checksum is $1 / 10^{12}$.
The user should adjust the number of checksum chunks in accordance with the amount being transferred.
For example, a user could type **btc@541290.852**/021 as the destination account ID when she is sending sending BTC 0.001 and **btc@541290.852**/021-288-184 when she is sending BTC 3.5 to the same account.

Users can be reasonably expected to make more mistakes when using Canonical IDs rather than Mnemonic IDs. Because of this, as a general rule, it is advisable to use longer checksums when using Canonical IDs.

2.3.8.1 Extended checksums

It is possible to build arbitrarily long checksums, by repeatedly hashing the last digest and extracting the next set of four checksum values, which are concatenated using dots, like in this example:

> **btc@bring-gentle.dish**/tissue-picture-country-process.**exotic-bracket-surge-traffic**.scene-alcohol-raw-girl
>
> **btc@461577.505**/907-657-196-686.**320-107-873-923**.770-024-714-393

This might make sense for extremely high-value transactions, but it seems overkill and unwieldy for most use cases, so it isn't a recommended practice.

2.4. Censorship

Censoring an *easypaysy* account per se is almost impossible, since the accounts themselves reside in the Bitcoin blockchain, which is widely available by multiple channels, even by satellite, and each payment goes to a different address, impossible to neither predict nor detect by a third party.

However, it can be quite easy to implement a blacklist feature, that protects users from sending a transaction to a known undesirable destination.

If the user specifies a Canonical ID or Mnemonic ID that has been blacklisted, her wallet software will readily warn her.
If she is using the Domain ID of an account that has been hijacked, the wallet software can detect the redirection to a blacklisted account's tx, and alert her of the risk.

Should any entity try to use this wallet feature to censor unwanted accounts, the user can have the option to either disregard the warning, load a different blacklist, or simply choose another wallet software.
Another censorship threat derives from the possibility of censoring not the account itself, but the rendezvous channels detailed in its Rendezvous descriptor. A powerful entity may be locally or globally capable of blocking access to an email account or MQTT server, for example. Having different protocols may alleviate this attack, but until a sufficiently robust protocol exists, this could remain an issue affecting interactive payments.

On the other hand, non-interactive payments are completely immune to censorship, as long as the user is capable of sending a regular Bitcoin transaction. This fact may discourage trying to censor interactive payments at all, to avoid pushing adversaries toward the much stealthier IOC payments (see *"2.6.1 Payment types"*).

2.4.1 Self censorship

The non-custodial nature of *easypaysy* accounts introduces new challenges, such as dealing with the loss of the keys used to create and maintain an account.

If the keys are stolen, but not lost, the owner of the account can and should revoke the account as soon as possible (see "2.5 *Account life cycle"*). If the keys are missing, the account owner loses the ability to update or revoke the account anymore. In that case, the best course of action may be to disown the account, by publicly announcing the situation by the same means previously used to announce the account ID. Aso, if the Rendezvous descriptor allows interactive payments, the contact protocols can be used to alert a potential payer of the issue.

Additionally, the owner of the account could try to include her own account in a black list by directly dealing with the list maintainers. She could prove her ownership by signing a message, in case she still has access to at least one of the two private keys (Identity or Value).

2.5 Account life cycle

An account can be in any of these five states (see figure 2, next):

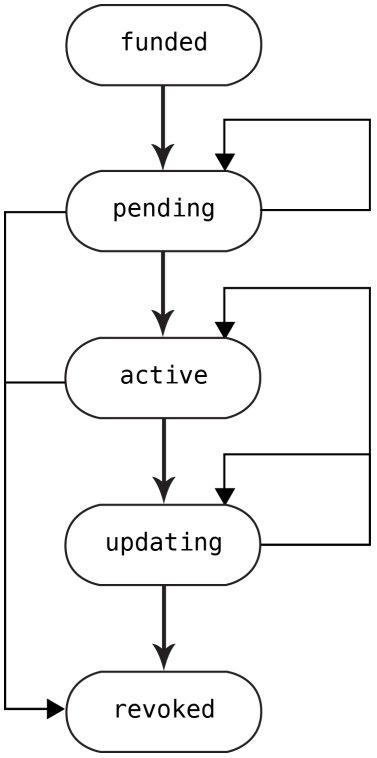

Fig. 2: State diagram of an account

a.- Funded: In this state, the account itself doesn't exist yet, but its associated 2-of-2 multisig address has already been funded by broadcasting a valid transaction that has an output pointing to it. The 2-of-2 multisig address is formed by building a multiple signature address that requires the signatures of both the Identity key and the Value key, in that order.

b.- Pending: An account is in this state when a transaction, in the proper *easypaysy* format, that spends the funds described in state a), has been broadcast to the blockchain but it hasn't reached maturity yet; that is, it isn't included yet in a block with 100 or more confirmations. This interval must be used by the account holder to test that the account has been properly set up and all the related rendezvous

services are fully operational. She could, for example, create an email account as indicated in the Rendezvous descriptor. If needed, she can also issue a remedial update transaction to fix any problems or errors she may have detected during this phase, in which case the counter for the 100 confirmations will be reset to zero.

c.- Active: The pending account enters this state after its container block has reached maturity, that is, it has at least 100 confirmations after entering the pending state or 6 confirmations after an update has been issued -see state d)-. Any funds lost or delayed due to a misconfiguration of an active account (like specifying a payment method the account holder isn't prepared or willing to process or wrongly specifying a rendezvous protocol endpoint) are the responsibility of the account owner.

d.- Updating: An account can be updated at any point in time after its activation, by spending its input with a new transaction that has an output with the same multisig address as the input and a second output that contains the updated Rendezvous descriptor. After activation, whenever an account is updated, it will enter the "Updating" state, until six confirmations have passed. This way, the account owner has the opportunity and time to fix an erroneous update before it becomes active. An update issued for an account currently in the Updating state will reset the counter to zero.

e.- Revoked: An account can be revoked at any time by spending it and sending its funds to a different address. Revocations take effect as soon as they are included in a block.

It is the responsibility of the potential payer to check the status of the destination account and act accordingly.

More specifically, the payer's software:

- Must either use a fully validating node or at least perform all the required SPV checks to verify the integrity of every transaction in the chain of updates leading to the most recent version of the account.
- Must refrain from sending payments to any account whose status is other than Active.
- Must follow the chain of updates (if any) and only use the information of the most recent update.
- Must fully comply with all the terms of the account's policy as described in the Rendezvous descriptor of the last update (see “2.6 *Rendezvous descriptor”*).

In figure 3, we can see a sample account, and the series of transactions used to create and change its state, until its final revocation.

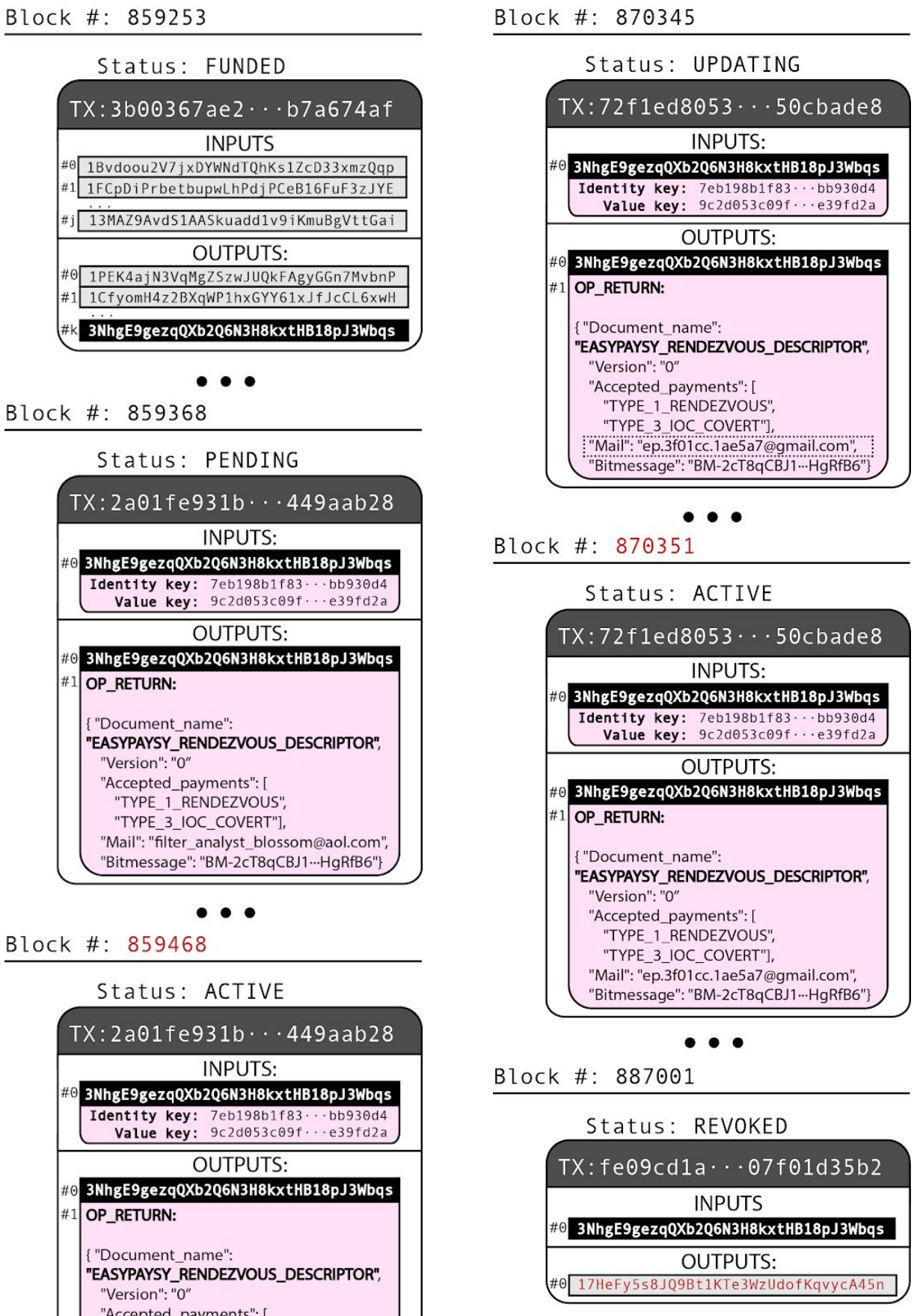

Fig. 3: Transaction history of an account

2.6 Rendezvous descriptor

In addition to the account ID, derived from its inclusion in a block, and the Identity and Value keys, that can be extracted from the account's transaction signature, the third main item of information of an account is the Rendezvous descriptor.
The Rendezvous descriptor is a JSON document, stored in the provably prunable output of the transaction, as the data field of the OP_RETURN operator.


```
{
   "Document_name":
"EASYPAYSY_RENDEZVOUS_DESCRIPTOR",
   "Version": "0",
   "Accepted_payments": [
       "TYPE_1_RENDEZVOUS",
      "TYPE_2_IOC_OVERT"
   ],
   "Mail": "fiber.burden.erupt@gmail.com",
   "Bitmessage": "BM-2cTZhb···NnZTjC69ke9BieU"
}
```


Fig. 4: Example of a Rendezvous descriptor

The Rendezvous descriptor is serialized using a highly optimized, dictionary-based, compression algorithm, before storing it into the OP_RETURN data field. The sample descriptor shown in figure 4 only takes 7 bytes in the current implementation of the protocol.

It includes several attributes:

1.- Document_name: A string that identifies this particular *easypaysy* document, with this fixed literal:
"**EASYPAYSY_RENDEZVOUS_DESCRIPTOR**"

2.- Version: A number that specifies the version of the Rendezvous descriptor (integer, starts with 0).

3.- Accepted_payments: A list of the payments that the account owner is willing to accept.

Possible payments are:

TYPE_0_UNSAFE_FIXED,
TYPE_1_RENDEZVOUS,
TYPE_2_IOC_OVERT,
TYPE_3_IOC_COVERT

The account must accept at least one of these payment types, though any non-empty combination is also valid.

2.6.1 Payment types

Here is a brief description of each payment type:

TYPE_0_UNSAFE_FIXED:
Type_0 payments must always use the same address, more specifically the address associated with the Value key. This is widely regarded as a bad practice, due to the total lack of privacy involved, thus its name. In fact, it is mainly included in the protocol so it can be given a discouraging name that dissuades its use (since blocking this type of payments is unfeasible). If possible, these payments should be avoided both by the account holder, by not listing it among the valid payment types, and by the payer, by using an alternative payment type, if available, or just refusing to make the payment altogether.

TYPE_1_RENDEZVOUS:
Rendezvous payments require interaction with the account in order to get a payment address for each payment.
If the "Accepted_payments" item includes this type of payment, the Rendezvous descriptor must list at least one contact protocol and its corresponding endpoint. In order to provide redundancy and protection against DOS attacks, the account can include more than one contact protocol and more than one endpoint for the same protocol.
(Other measures against DOS attacks include the requirement to provide hashcash with the payment request, forcing its request to come

signed from a valid *easypaysy* account, or even requiring a token payment via LN to get a payment address. These and other measures could be implemented on a case by case basis and in response to actual attacks to the account, as they begin to happen. In any case, Hollywood payments -Type_2 and Type_3- are immune to DOS attacks.).

In the previous example (see figure 4) the account accepts two rendezvous mechanisms, "Email" and "Bitmessage" with their corresponding endpoints.
At the time of writing this, there are up to four communication mechanisms planned as rendezvous protocols, namely:

Https, email, Bitmessage and MQTT.

Note: The Rendezvous compression dictionary has reserved a token for a possible fifth protocol named "*Easypaysy*" designed specifically for this task, but that is out of the scope of current efforts.

TYPE_2_IOC_OVERT, TYPE_3_IOC_COVERT:
Both of these payments are non-interactive. As such, they do not require additional entries in the Rendezvous descriptor to specify a communication protocol and endpoint.

Inversion Of Control payments (a.k.a. "Hollywood payments") invert the control of the payment process so that it is the payer, not the payee the one choosing the destination address of each payment.

The mechanisms needed to enable this type of payments are described in a section below (see "*3.2.1 IOC payments"*).

For now, suffice it to say that the only difference between a Type_2 and Type_3 payment is that the former is way more easily detectable as a Hollywood payment than the latter.

2.6.2 Compressing the Rendezvous descriptor

Space in the Bitcoin blockchain is a scarce resource that many argue has been misused, particularly in the past. The 0.9.0 release of Bitcoin Core added support for a new script function, OP_RETURN. The release notes[7] read: "*This change is not an endorsement of storing data in the blockchain. The OP_RETURN change creates a provably-prunable output, to avoid data storage schemes – some of which were already deployed – that were storing arbitrary data such as images as forever-unspendable TX outputs, bloating bitcoin's UTXO database.*"

Whereas there is no actual limit on how much data can be stored within an OP_RETURN output, Bitcoin core nodes do not relay more than 80 bytes, so the protocol abides by this limit. The *easypaysy* protocol takes advantage of this OP_RETURN operator in a most respectful way, striving to make compatible the need to store some information with a very efficient compression algorithm in order to minimize the externality costs.

While the decision to use JSON over a proprietary or binary format may seem at odds with that intent, the compression levels achieved, in excess of 95% in many cases, makes us believe this approach represents a valid tradeoff between ease of use and efficiency.

Two main techniques are used to attain this very high level of compression, namely, using a dictionary-based compression algorithm and using variables that are expanded at run time.

[7] Bitcoin Core 0.9.0 release notes

2.6.2.1 Dictionary compression

The dictionary itself is a very simple JSON document with a series of attribute/value pairs. Each attribute is a token that represents a literal to be compressed. Tokens are expressed in hexadecimal, as shown in table 2. Each token can occupy one or more bytes. Single byte tokens use the hexadecimal range:
[0x00..0x1E] ∪ [0x80..0xFE].
Multiple-byte tokens begin with 0x1F. When decoding a compressed Rendezvous descriptor stored within the OP_RETURN output, the decompression algorithm replaces every token with its associated string. The remaining bytes, whose value is in the range [0x20..0x7F], are transcribed verbatim.

The dictionary is manually crafted taking into account both the length of the expanded literal, and its expected frequency.

```
...
    "91": "<payments_mask:!>",
    "92": "<payments_mask:\">",
    "93": "<payments_mask:#>",
    "94": "<payments_mask:$>",
    "95": "<payments_mask:%>",
    "96": "<payments_mask:&>",
    "97": "<payments_mask:'>",
    "98": "<payments_mask:(>",
    "99": "<payments_mask:)>",
    "9A": "<payments_mask:*>",
    "9B": "<payments_mask:+>",
    "9C": "<payments_mask:,>",
    "9D": "<payments_mask:->",
    "9E": "<payments_mask:.>",
    "9F": "<payments_mask:/>",
...
```

Table 2.- Sample token entries in the compression dictionary

In the current version of the prototype implementation, the majority of tokens only take one byte, even though some values are still unused. For example, as we can see in Table 2, tokens 0x91 through 0x9F are assigned to the literals that store the variable <payments_mask:?>, used to represent the combination of types of payments accepted by the account.

2.6.2.2 Rendezvous variables

The second strategy employed to save space when serializing the rendezvous descriptor is the use of variables. These variables are evaluated at run time, converting a compact placeholder into a notably longer string.

For instance, the variable `userid(-)`, which only takes one byte after begin tokenized by the compression algorithm, could be evaluated into something like:

```
river-congress-tattoo
```

There are variables to represent the main attributes of an account such as its Canonical ID, Mnemonic ID, TXID, Identity and Value public key and address, etc.

3 Payments

Easypaysy payments can be divided into interactive and non-interactive payments.

3.1 Interactive payments

Interactive payments (Type_1), as their name denotes, require an interaction between the sender and the receiver of a payment before the transaction can be prepared.

The purpose of this interaction is twofold:

First, it lets the payee decide and communicate the payer what address to use for each payment.

Second, because the address provided for the payment is signed with the Identity key of the recipient account, the payer can prove that he paid to a sanctioned address.

As we can see in figure 5, to make a push payment the payer must first interact with the blockchain, in order to retrieve the information associated with the account he wants to pay to, then with the payee to get a payment address and its signature, and finally with the blockchain again to broadcast the transaction.

The detailed sequence of events is as follows:

1.- The payer retrieves the account information from the blockchain.
2.- The payer then sends a JSON document to the recipient -named "ROE_REQUEST"- asking for her Rules of Engagement. These are the conditions the recipient demands to supply a new payment address. More specifically, the recipient can ask for a certain amount of hashcash, to combat spammers.
3.- The recipient sends back her answer using another JSON document, named "ROE_REPLY".

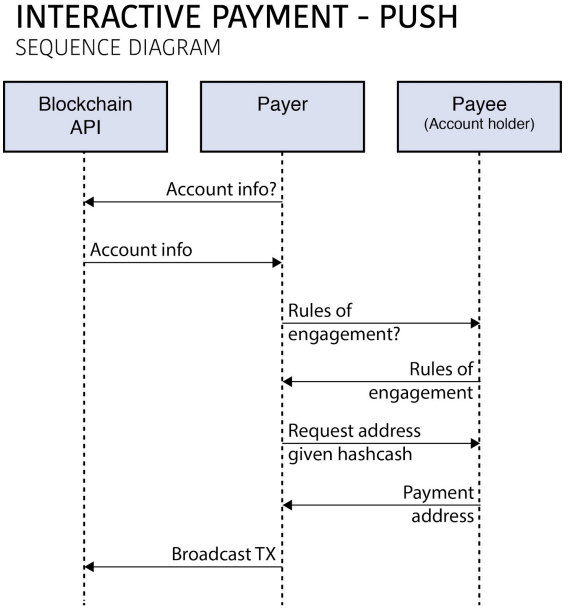

Fig. 5: Sequence diagram of push interactive payments

4.- The payer sends a "PAYMENT_REQUEST" document, whose attributes include: amount_to_pay, hashcash_stamp, payment label, reply_to address, ...
5.- The payee will then send the "PAYMENT_REPLY" a JSON document with the requested address. This document includes a non-mandatory "Pay_to_invoice" attribute, that the payee can fill in with a Lightning Network invoice for the amount requested. If given the option, the payer will then decide whether to pay on-chain or using the Lightning Network.

The "PAYMENT_REPLY" document also includes an attribute named "Pay_to_address_signature" and another named "Pay_to_invoice_signature" where the payee includes the corresponding signatures. These attributes confer the non-repudiability characteristic to these payments.
6.- The payer will either broadcast a transaction to the blockchain or pay using the Lightning Network (when given the option).

3.1.1 Encryption

All the communications between payer and payee must be encrypted using the ECIES protocol. This security protocol relies on a ECDH exchange to share an AES key, which is then used to encrypt and decrypt the JSON documents described above.

Even if the communication channel is compromised, the attacker will also need access to the private Identity key of the account in order to be able to supplant the account owner.

3.1.2 Perfect forward secrecy

To ensure perfect forward secrecy, all the interactions -except for the first one, that uses the Identity key of the recipient's account- must use an ephemeral -preferably non-deterministic- public key. This way, even if the Identity key of the account is ever compromised, the privacy of past or future interactions between the payer and the payee is preserved. To that end, every request includes an attribute named "Encrypt_answer_with_public_key" where each party communicates the other the public key to be used to encrypt the next message.

3.1.3 Pull payments[8]

All typical Bitcoin payments are push payments. That means that it is always the payer who initiates the sequence of events that ends with a payment.

Pull payments, on the other hand, allows one party -usually a business- to withdraw or 'pull' funds from the other party -usually a customer-. We can see their sequence diagram in figure 6.

[8] Pull payments are planned for a future release of the protocol.

They are typically used when there is a contractual relationship between a company and its customers, such as with phone operators, cable tv providers, etc.

When both payer and payee have an *easypaysy* account that supports interactive payments (Type_1), it is possible to set up pull payments between them.
Such a setting, which seems particularly well suited to a dedicated secure hardware device, could enable one party to authorize a series of recurring payments from another, within a given set of rules.

One customer could, for instance, authorize her ISP to invoice her for up to a maximum monthly amount. She can do this by inserting a new rule within her secure device that identifies the ISP's *easypaysy* ID, and the maximum monthly amount she is willing to pay automatically.

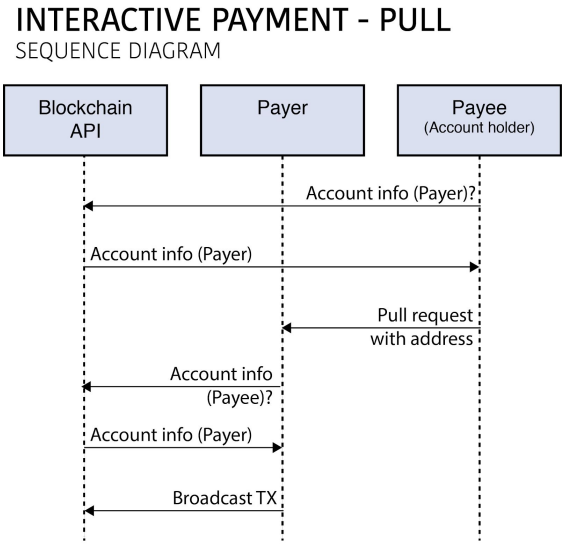

Fig. 6: Sequence diagram of interactive payments - Pull

Every month, her ISP will send a JSON document, named "PULL_REQUEST" and signed with its Identity key, requesting the payment for the past month's service. Upon receipt, the user's device will verify the signature of the request, matching it against the

source *easypaysy* account and validate whatever rules she defined -most importantly the due date and the amount-. If everything matches the given set of rules, it will then issue a payment transaction to the given address, that came encrypted and signed within the pull request.

3.2 Non-interactive payments

These payments present a challenge over their interactive counterparts, since they require the sender of the payment to come up with a different and unpredictable address for each new payment, without any interaction with the account holder.
In figure 7, we can see the sequence diagram of these payments.

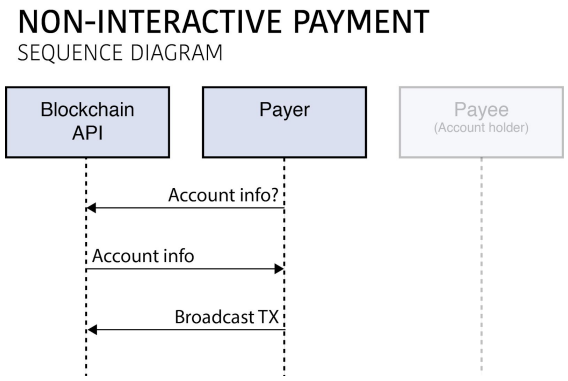

Fig. 7: Sequence diagram of non-interactive payments

In order to enable non-interactive payments, the *easypaysy* protocol defines a new type of payments, named IOC payments (Inversion Of Control) or, more informally, Hollywood payments.

In contrast with regular Bitcoin payments, where the recipient first creates a private/public key pair and then communicates the payer its associated address, IOC payments invert the control over this process so that it is the sender of the payment, not the recipient, who chooses a new payment address for each transaction.
This inversion of control presents a series of challenges, as we see below:

a) The payer must be able to select a payment address of which he is certain the recipient will have the corresponding private key.
b) The payee must be able to detect that she has been sent an IOC payment, without first getting in contact with the payer, just by observing the flow of new transactions.
c) The payee must be certain that the payer doesn't know the private key of the address used in the payment.
d) The addresses must be selected in a way that ensures only the payer and payee themselves will be able to link this payment.
e) The address must be selected in a way so that the payee can't repudiate the payment.

IOC payments, as we can see below, fulfill all of these requirements.

3.2.1. IOC payments

{1} Let Alice and Bob be two Bitcoin users, so that Bob wants to make one (or more) on-chain payment(s) to Alice.

{2} Alice publishes a transaction in the blockchain that we'll call an *easypaysy* root transaction (ep_root_tx). This transaction is the *easypaysy* account of Alice and <a,A> is her Value key.

Alice knows a and A, where:

{2.1} a ; a private key that Alice creates by any secure means.

{2.2} A = a * G ; the public key associated with the private key <a>.

Please note that Alice signs <ep_root_tx> using <a>, exposing <A> in the process, but keeping <a> secret for herself .

{3} Bob has knowledge of Alice's account ID through any channel (website, company's stationery, business card, email...). Bob parses Alice's account ID and retrieves her *easypaysy* account information from the <ep_root_tx> transaction.

{4} Bob creates a secure private/public pair of keys, unique for this payment:

```
{4.1} b            ; Bob's private key
{4.2} B = b * G    ; Bob's public key
```

{5} Bob calculates a scalar <n> and its corresponding point in the Secp256k1 elliptic curve <N>, so that:

```
{5.1} n = digest(sha256(b*A))
        = digest(sha256(b*a*G))
{5.2} N = n * G
```

{6} Bob calculates <C>, another point in the elliptic curve, so that:

{6.1} `c = a + n;` (Note that, for now, neither Bob nor Alice can calculate <c>, since Bob doesn't know <a> and Alice doesn't know <n>)

```
{6.2} C = (a + n) * G
        = (a * G) + (n * G)

                  ; a * G = A {2.2}
                  ; n * G = N {5.2}
```

{6.3} `C = A + N ;` Bob knows A because of {3} and N because of {5.2}.

{7} Bob derives <C_address>, the bitcoin address corresponding to <C>, and creates a payment transaction for Alice that has (at least) two outputs.

Bob broadcasts this transaction and (optionally) notifies Alice of its WTXID.

output #0:

```
Amount: any
Script: standard p2pkh,
        using <C_address>
```

output #1:

```
Amount: 0
Script: OP_RETURN DATA
```

(where DATA = <B>, as indicated in {4})

{8} Upon receiving notification from Bob in {7}, or just by monitoring the flow of new transactions, Alice:

{8.1} Retrieves <B> from the <OP_RETURN data> field of the <output #1> of the TX created and published in {7}

{8.2} Calculates n, c and C:

```
n = digest(sha256(a*B)); {4.2}
  = digest(sha256(a*b*G))
  = digest(sha256(b*a*G));

   ;(same <n> as in {5.1})

c = a + n

   ;(same <c> as in {6.1})

C = c * G
```

{9} Having computed <c> and <C> in {8.2}, Alice verifies that she is in control of the funds locked in the <output #1> of the transaction that Bob sent her in {7}.

{10} In practice, the protocol defines a deterministic way to compute a new value of <n> for each payment. As a result, future payments from Bob to Alice don't need to include a public point within an OP_RETURN output.

```
{10.1} n = digest(sha256(i&b*A))
                  ; i is the index
                    of the payment
                    from Bob to
                    Alice
```

3.2.1.1 Non-repudiability of IOC payments

Interactive payments derive their non-repudiable nature from the fact that, during the interaction between payer and payee, the payee must sign the prescribed payment address with her Identity key.

That signature is lacking in the case of Hollywood payments, so the payer has to resort to a different way to disqualify any attempt of repudiation.

He needs to prove that:

a) He sent an IOC transaction to the proper *easypaysy* account, following all the prescriptions of the protocol.
b) The account was active at the time the payment was sent, and its Rendezvous descriptor allowed the specific type of payment he used (Type_3 or Type_4).
c) He provided a valid point <B> in the Secp256k1 elliptic curve within the OP_RETURN data field of the same transaction, encoded in the right format (see point {4} of the algorithm).
d) At least one of the outputs of his transaction is sent to the address, associated to the point described in c).
e) The transaction has been successfully inserted in the blockchain, and has enough confirmations to be considered immutable.

In order to prove all of that, the payer will simply need to disclose both the WTXID of the transaction and the value of <b> from which he derived <B>, included in the OP_RETURN field.

Thus, by following the algorithm described before, any impartial observer can attest that the payer received a valid payment of which she has enough information to unlock it.
Since the payer uses a different, non-predictable value of <b> for each transaction, by revealing it he will only compromise the privacy of that particular payment.

3.2.2 Overt vs Covert payments

As we have just seen in the previous point, the first IOC payment from a sender to a particular account, requires some information, encoded within an OP_RETURN output, that the recipient will use both to detect that she is the intended destination and to gather enough information to compute the corresponding private key.

The minimum information needed to that end is the value of the public key B (see "*3.2.1 IOC payments"*) that the sender is using for that particular payment.

3.2.3 Uber-compressed format

Easypaysy defines a custom format for storing a public key, named *uber-compressed*, which builds upon the traditional compressed format for ECDSA keys. Instead of taking 33 bytes, it has a variable size that goes from a maximum of 31 bytes to a minimum of 28.

This is the format to be used within the OP_RETURN data field of the first payment to a particular *easypaysy* account.

In order to strip those extra bytes from the public key, the format mandates that:

a) The header byte, the one that indicates the parity of the Y coordinate of the public key, is not included.
b) The leftmost byte of the public key, that must always be zero, is also

stripped from the public key representation.

c) If the next up to 3 bytes are zero (this is optional) they must be removed from the representation.

Due to the requirement expressed in b) when selecting a public key for the payment, the sender must ensure that the first byte is zero. This is easily accomplished by iterating a nonce to derive <B> until it fulfills the condition.

The computational effort needed to find a public key whose first byte is zero -128 attempts on average- is negligible when using any modern CPU.

Thus, it seems reasonable to impose this requirement to save one byte. The sender can then choose to spend more CPU time in order to find up to three more zero bytes, or stick to the required minimum of one.

The recipient will have to perform up to six attempts to reconstruct the key that comes in the OP_RETURN output[9], with an average of less than three, since shorter paddings will probably more common, at least in the near future.

Overt payments must add the prefix bytes "EP" (hex 4550) to the public key, while covert payments mustn't.

This is in the only difference between overt and covert payments.

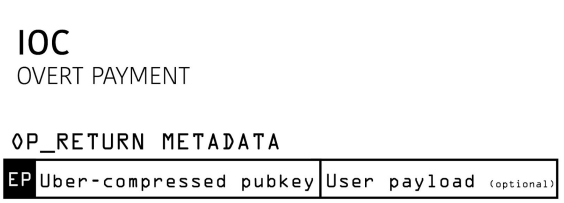

Fig. 8: Structure of the OP_RETURN data field of an overt payment

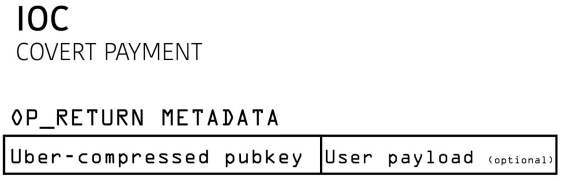

Fig. 9: Structure of the OP_RETURN data field of a covert payment

In practice, it means that covert payments are somewhat cheaper for the sender -since they always take two bytes less than their overt counterparts- while being a little more taxing for the recipient, since she has to scan and test every transaction that has an OP_RETURN output with a length longer than 27 bytes.

Since the user is allowed to include custom information right after the public key, it will be very difficult for an observer to know for sure that a covert payment is an IOC payment. This comes at the cost of the recipient having to analyze more transactions than in the case of overt payments.

[9] Since it is permissible to include some user data right after the uber-compressed public key, she can't know for sure its actual size without trying out the different possibilities.

3.3 Bandwidth

The bandwidth that a particular user needs to process every potential IOC transaction within a block, in order to detect which transactions are addressed to her, can be estimated using this formula:

A * F * S, where:

A = Average number of transactions per block.

F = Fraction of transactions with an OP_RETURN potentially hosting an IOC payment (size >= 28 bytes).

S = Average size of the OP_RETURN output of potential IOC transactions.

The average size of the OP_RETURN metadata is 23.4 bytes [4], with about 1.16% of transactions containing an OP_RETURN output.

We are going to consider 100% of that 1.16% as potential transactions that need to be scanned.

To that value, we need to add the expected number of IOC transactions that have an OP_RETURN data field attached. Only the first IOC transaction between any two given *easypaysy* users need to contain an OP_RETURN output.

For lack for any empirical data on the use of *easypaysy*, we will make these pretty extreme assumptions:

- We'll use 2500 as the average number of transactions per block.
- 90 % of all the transactions in a block are *easypaysy* transactions.
- 90 % of those transactions are IOC transactions.
- 90 % of those IOC transactions include an OP_RETURN data field.
- The value of S is 40 bytes

We can then estimate:

A = 2500
F = 0.016 + 0.9 * 0.9 * 0.9 = 0.745
S = 40

Then, A * F * S = 74500 bytes per block.

So, provided a user has access to a beacon node that relays all OP_RETURN data fields (we are counting here all of them, including the non-candidates, to provide better privacy) the required data speed is 74500 bytes/block * 8 bits/byte * 6 blocks/hour * 3600 secs/hour = 993.3 bits/second, well below the max speed of even GPRS[10].

As for the total monthly data transfer, we can estimate 365.25 / 12 days/month * 144 blocks/day * 73060 bytes/block = 300,221,980 bytes/month or about 286.6 MB/month.

10 https://en.wikipedia.org/wiki/General_Packet_Radio_Service#Coding_schemes_and_speeds

4 Scalability

Bitcoin has currently a maximum throughput of about 7 transactions per second [3], with no credible plans to increase that on-chain capacity in the near future, if at all. Since each *easypaysy* account uses at least one transaction, this places a hard limit on how many accounts can be open in any given time frame.

Easypaysy can only be really helpful if it ever becomes widely used, so it pays to analyze the feasibility of onboarding a large number of users, and the impact that would have on the blockchain.

There is no good data to measure the actual number of Bitcoin users, with some known estimates varying wildly between 13.2 million [11] and 40 million[12].

Just taking the lower of these two estimates, assuming that 10% of all the transactions in a block are used to create new *easypaysy* accounts (not counting the funding transactions, since a single transaction can fund multiple accounts), we can estimate about 0.1 * 2500 transactions / block = 250 new accounts per block, or about 250 * 144 blocks/day = 36000 new accounts per day or about 13.15 million new accounts per year, close to the lower estimate of the number of users.

While it is clearly possible to onboard such a number of users every year, that's still a small percentage of the world population.

In consequence, it seems desirable to find mechanisms to alleviate this load on the blockchain, both to facilitate the creation of a large number of accounts and to keep the the associated costs within a reasonable limit.

To that end, we envision two possible mechanisms: surrogate accounts and master accounts.

4.1 Surrogate accounts[13]

A surrogate account is an *easypaysy* account that is created in one blockchain but intended to be used for payments in a different blockchain.
The generic format of a surrogate account ID is:

<account_id>[:<surrogate_index>]

Where <surrogate_index> is the index of the surrogate blockchain, as defined in the SLIP_0044[14] standard, and [] implies optionality.
An example of surrogate account ID, hosted in the Litecoin blockchain could be this:

btc@558470.886/931-486**:2**

Conversely, a Litecoin account, hosted in the Bitcoin blockchain could be like this:

ltc@830209.82/319-003**:1**

Surrogate accounts let Bitcoin users benefit from the lower fees of the Litecoin blockchain while Litecoin users can benefit from the higher security of the Bitcoin blockchain.

[11] https://medium.com/@coventureresearch/how-many-people-own-bitcoin-9dd3ddd7bba5

[12] https://www.statista.com/statistics/647374/worldwide-blockchain-wallet-users/

[13] This is a planned feature for a future release of the protocol.

[14] See: https://github.com/satoshilabs/slips/blob/master/slip-0044.m.

4.1.1 Implicit surrogacy

The protocol establishes an implicit surrogacy between the Bitcoin and Litecoin blockchain. This allows for a simpler account ID, that hides the complexity of the surrogation process from the end user.

Because of that, the two previous example account IDs could be written simply as:

btc@558470.886/931-486
and
ltc@830209.82/319-003

4.2 Master accounts

While the surrogacy of accounts transfers the burden from one blockchain to another, it doesn't completely solve the scaling problem.

Master accounts are another mechanism, also planned for a future release of the *easypaysy* protocol, that enables a scaling of three orders of magnitude.

A master account is an *easypaysy* account that enables the creation of multiple individual accounts (up to 2048) within a single blockchain transaction. This way, using the same assumptions as in the previous point, up to 13.15 million accounts * 2048, or about 27 billion new accounts, could be easily created every year, more than enough to accomodate the whole world population.

4.2.1 Master account IDs

Since the main raison d'être of *easypaysy* is to promote ease of use, it is critical that the implementation details are hidden from the user. As a result, the format of master account IDs is designed to mimic that of single-account IDs, no matter what the underlying differences may be.

The ID of a master account follows this structure:

btc@master_idx.slave_id/checksum

Below we can see its constituent parts:

master_idx is a value that represents the order in which a specific master account has been created in the blockchain. That is, the first master account ever created will be 0, the next will be 1, and so on.

slave_id is a value that points to one individual (slave) account within the master account. It encapsulates two items, namely:

- **slave_idx**, a value that specifies the individual account within the master account. It takes values from 0 to 2047.
- **slave_chk**, the checksum that protects the integrity of **slave_idx** . Its value is calculated with this formula:

  slave_chk =
  (slave_idx + checksum_chunk_0) % 2048 .

  For instance, when **slave_idx** = 1345, and **checksum_chunk_0** = **847:**

  slave_chk = (1345+847) % 2408 = 642.

These two items are combined into a single value, applying this formula:

slave_id = 2048 * **slave_idx** + **slave_chk**

Following this example:

slave_id = 2048 * **1345**+ **642**
= **2755202**

Thus, we could write the Canonical ID of the slave account #1345 of the master account #9005 as follows:

btc@9005.2755202/847-967-108-553

And this could be its Mnemonic ID:

btc@able-cliff.popular-expect/stable-vault-brand-medal

4.2.1.1 Parsing the account ID

Every *easypaysy* ID follows this format:

btc@block_or_master_id.tx_or_slave_id/checksum

where:

block_or_master_id can either point to the blockchain block containing a standard account, or be the ordinal pointing to a master account and **tx_or_slave_id** can either point to the index of the a standard account within a block or be the ordinal of a slave account within a master account.

Due to this (intentional) ambiguity, given an *easypaysy* ID, the parser must evaluate both possibilities and opt for the one that is valid.

In case of a collision between a standard and a master account ID, the checksum must be used to disambiguate the parsing.

Note: In order to avoid confusion, if the collision extends up to the first chunk of the checksum, the master account will be invalidated and the parser will return the regular account instead.

4.2.2 Master account metadata

Master account TXs must have an OP_RETURN output, containing a JSON document, named "**EASYPAYSY_MASTER_ACCOUNT_DESCRIPTOR**", that includes two items of information, namely:

"**Authoritative_server_url**"
and
"**Merkle_root**".

The first item, "**Authoritative_server_url**" points to the url of the authoritative server, that the end user can use to request the information of a particular account, such as:

"Authoritative_server_url":
"https://example.com/easypaysy/master/btc"

The request for a specific slave account, will append the value of the intended account ID, like in:

http://example.com/easypaysy/master/btc/**account/9005.2755202**

For added privacy, the request can omit the slave_id part, thus requesting the whole set of slave accounts within the given master account, as in:

http://example.com/easypaysy/master/btc/**account/9005**

Upon request, the server will return a JSON document containing the Rendezvous descriptor for one particular account (including its Identity key and Value key) or for all the slave accounts, in case the request didn't specify the slave_id.

The user will then issue a second request to get the Merkle proof:

http://example.com/easypaysy/master/btc/**merkle_proof/9005.2755202**

Again, omitting the slave_id will result in the server returning the whole Merkle tree of the master account, as in this request:

http://example.com/easypaysy/master/btc/**merkle_proof/9005**

The user can then calculate the SHA256 digest of the JSON document received in the first request and verify that it matches the Merkle proof received in the second request, in accordance with the Merkle root included in the "Merkle_root" attribute within the EASYPAYSY_MASTER_ACCOUNT_DESCRIPTOR document.

The Merkle root will be calculated in a similar manner to the Merkle root of a Bitcoin block, after sorting all of the accounts in ascending order of their corresponding SHA256 hash digests.

4.2.3 Mirror servers

The risk that the authoritative server designated within the EASYPAYSY_MASTER_ACCOUNT_DESCRIPTOR could become unavailable can be mitigated with the use of mirror servers.

Since the integrity of each individual account is protected by its hash and the corresponding Merkle proof, there is no additional risk in using a mirror instead of the authoritative server of a master account.
As we have seen before, for added privacy, -especially when dealing with a mirror server- a user can request the information of all of the slave accounts within a particular master account at once, at the expense of a few extra kilobytes. Even without compression, 2048 accounts should occupy less than 1 megabyte.

It is conceivable that the mirror could charge for this service, perhaps requiring a small LN payment per request, so there will be an economic incentive to preserve the information associated with every master account ever published into the blockchain.

The process to create a Master account requires some coordination to aggregate the individual slave accounts. All of these details, including the definition of the adequate mechanisms to handle the life cycle of each slave account are to be defined in the future.

5 Further work

We have presented, grosso modo, the current status of the protocol. Some, easy to implement, and potentially useful features, such as chargebacks, have intentionally been left out.

They will be added to the protocol in its due time, probably along with some still unforeseen capabilities, after we receive enough feedback and suggestions from the community. Once we have a clear vision of the needs and priorities of the different actors of the ecosystem (users, wallet developers, hardware makers, etc.), we will freeze the first set of specifications and publish it along with a preliminary roadmap at www.easypaysy.org.

In the meantime, during the testing phase, in order to aid block parsers to discriminate the different types of *easypaysy* accounts, the OP_RETURN metadata should include a prefix, indicating the kind of account represented by the transaction. They are especially critical to identify Master accounts, since their ID is of an ordinal rather than positional nature.

During the testing phase, they will take these names and values:

MAGIC_EP_STANDARD_T: 0x0000000001
MAGIC_EP_SURROGATE_T: 0x0000000002
MAGIC_EP_MASTER_T: 0x0000000003

The definitive set of MAGIC words will probably have the same length, but different values.

6 Conclusions

We proposed *easypaysy*, a layer-two protocol designed to fix some of the biggest UX problems that Bitcoin users face nowadays.

By implementing non-custodial accounts directly on the blockchain, Bitcoin users have access to user-friendly account IDs, similar to current email addresses or bank numbers. This way, the dreaded bitcoin addresses can be totally hidden from the user experience, much like word wide web users never have to see an ip address.

We have shown that, through the use of both interactive and non-interactive modes of operation, *easypaysy* payments can cover many different scenarios, while providing similar or even greater levels of privacy and safety than currently possible.

We have also seen that it is possible to enact mechanisms to create and maintain a very large number of accounts, with a relatively low impact in the blockchain.

Finally, although *easypaysy* is meant to be primarily a system to greatly improve the user experience, we have shown that it can also give support to additional features, such as pull payments, difficult or impossible to implement without the support of an open infrastructure of pseudonymous, non-custodial accounts.

References

[1] Wuille, Pieter & Maxwell, Gregory, **"Base32 address format for native v0-16 witness outputs"** 2017. https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki

[2] S. Nakamoto, **"Bitcoin: A peer-to-peer electronic cash system,"** 2008. http://bitcoin.org/bitcoin.pdf

[3] Croman, Kyle & Decker, Christian & Eyal, Ittay & Gencer, Adem Efe & Juels, Ari & Kosba, Ahmed & Miller, Andrew & Saxena, Prateek & Shi, Elaine & Sirer, Emin & Song, Dawn & Wattenhofer, Roger. (2016). **"On Scaling Decentralized Blockchains"**. Bitcoin and Blockchain. 9604. 106-125. 10.1007/978-3-662-53357-4_8, http://fc16.ifca.ai/bitcoin/papers/CDE+16.pdf

[4] Bartoletti, Massimo, and Livio Pompianu. "**An empirical analysis of smart contracts: platforms, applications, and design patterns."** *International conference on financial cryptography and data security*. https://arxiv.org/pdf/1703.06322.pdf)