Flex Protocol Draft Version 0.2.0

Peer Node

November, 2015

Abstract

This document outlines the protocol used by the Flex network to mine credits and trade cryptocurrencies as well as create, transfer and withdraw from deposit addresses whose private key is distributed throughout the network.

Contents

1 Introduction

Flex builds on the bitcoin code by adding additional messages and message handlers. This document outlines the data structures and messages used for the cryptocurrency, cryptocurrency trading, and deposit parts of Flex.

2 The System of Credits used in Flex

Credits are either created as fees to pay relays, or are mined into existence, or are outputs of transactions.

Each mined credit corresponds to the concept of a "block" in other cryptocurrencies, but with a subtle difference. Those currencies group transactions together into blocks. Flex groups credits together into batches.

In other existing cryptocurrencies, each miner includes a "coinbase transaction" in each block, which pays the reward for mining to the miner.

Credits in Flex work slightly differently:

- Each miner groups new credits into a batch of credits and calculates the root of that batch.
- The miner combines this with other information needed for mining, including the miner's public key.
- The miner hashes the information and computes a proof of work from the hash.
- The miner has now mined a credit and proceeds to publish it.
- The newly-mined credit will be included in the next batch.

3 Elementary Data Types

The table below shows the most common data types used in messages. The native data types, uint64_t, uint32_t and uint8_t are also used.

Elementary Data Types			
Data Type	**Description**	**Use**	**size**
vch_t	std::vector<unsigned char>	data	variable
uint160	unsigned 160-bit integer	hash, measure of work	20 bytes
uint256	unsigned 256-bit integer	hash, private key, secret	32 bytes
Point	Elliptic Curve Point	public key, commitment to secret	34 bytes
CBigNum	OpenSSL big integer	private key, secret	variable

The 34 bytes in the Point class consist of 1 byte to specify the curve, 1 byte to encode a parity bit, and 32 bytes to encode one of the point's two coordinates.

4 Composite Data Types

The humble credit is the simplest composite data type used:

Credit		
Field Name	**Data Type**	**size in bytes**
amount	uint64_t	8
keydata	vch_t	20 or 34

The keydata field contains either a public key (34 bytes) or a key hash (20 bytes).

If the credit has been included in a batch of credits, then it acquires additional metadata:

CreditInBatch		
Field Name	**Data Type**	**size in bytes**
amount	uint64_t	8
keydata	vch_t	20 or 34
position	uint64_t	8
branch	std::vector<uint160>	variable
diurn_branch	std::vector<uint160>	variable

The branch variable is a list of hashes which link the hash of the (keydata, amount, position) or (credit, position) combination to the batch root.

The diurn branch variable connects the batch root to the root of a collection of batches called a diurn (described later).

CreditBatch		
Field Name	**Data Type**	**size in bytes**
previous_credit_hash	uint160	20
offset	uint64_t	8
credits	std::vector<Credit>	variable

A CreditBatch is an object to which we can add Credits one by one and which assigns numerical positions to them starting at the specified offset.

We can then ask it for:

- A CreditInBatch object for any credit which was added. This can serve as a proof that the (credit, position) combination was in the batch.
- The batch root (uint160), which is the information needed to validate any of the CreditInBatch objects from this batch.

If we have (or might at some future time acquire) the private key, we will add the CreditInBatch object to our Wallet.

Wallet		
Field Name	**Data Type**	**size in bytes**
seed	uint256	20
credits	std::vector<CreditInBatch>	variable
spent	std::vector<CreditInBatch>	variable

The seed in the wallet is a hash of a password and is used to deterministically generate addresses. There are no accounts, just addresses. Keys can be imported as well, in which case the private keys are saved in the encrypted database.

The spent CreditInBatch vector is there so that in the event that we need to remove batches from the tip of the block chain, we can recover the credits that were spent in the meantime.

The wallet also generates transactions:

UnsignedTransaction		
Field Name	**Data Type**	**size in bytes**
inputs	std::vector<CreditInBatch>	variable
outputs	std::vector<Credit>	variable
pubkeys	std::vector<Point>	variable

The inputs are taken from the credits in our wallet and the outputs are just raw credits that aren't in a batch yet. The public keys are there for those CreditInBatch objects which have a key hash instead of a public key in their keydata.

We sign the UnsignedTransaction to get a SignedTransaction:

SignedTransaction		
Field Name	**Data Type**	**size in bytes**
rawtx	UnsignedTransaction	variable
signature	Signature	64

The signature looks like this:

Signature		
Field Name	**Data Type**	**size in bytes**
exhibit	uint256	32
signature	uint256	32

where the exhibit and signature fields are the variables e and s from a so-called Schnorr signature.

The above data structures define the monetary system apart from the mining aspect of it. A credit which has been mined contains information about the state of the network encoded in it:

MinedCredit		
Field Name	**Data Type**	**size in bytes**
amount	uint64_t	8
keydata	vch_t	34
offset	uint64_t	8
previous_mined_credit_hash	uint160	20
previous_diurn_root	uint160	20
diurnal_block_root	uint160	20
hash_list_hash	uint160	20
batch_root	uint160	20
spent_chain_hash	uint160	20
relay_state_hash	uint160	20
total_work	uint160	20
difficulty	uint160	20
diurnal_difficulty	uint160	20
timestamp	uint64_t	8

The spent chain is a list of 1's and 0's which tell us which positions (those assigned to credits by the CreditBatch) correspond to spent credits. With each new batch of size N, N new 0's are appended to the spent chain, and a number of earlier bits in the spent chain will be flipped from 0 to 1 because they're the inputs of transactions.

The relay state is a collection of data relating to relays, namely:

RelayState		
Field Name	**Data Type**	**size in bytes**
latest_credit_hash	uint160	20
latest_relay_number	uint32_t	4
relays	std::map<Point, uint32_t>	variable
line_of_succession	std::map<Point, Point>	variable
actual_successions	std::map<Point, Point>	variable
subthreshold_succession_msgs	std::map<Point, std::set<uint160>>	variable
disqualifications	std::map<Point, std::set<uint160>>	variable
join_hash_to_relay	std::map<uint160, Point>	variable

If we know the relay state, then we can determine:

- Who the relays are (i.e. their public keys)
- What order they joined in (the "relays" variable maps pubkey to join order)
- Who is the successor for each relay
- Which relays have actually gone offline, leaving their duties to their successors
- Which duties will be inherited if more succession messages are sent to the successor. For example, a relay might not respond for a while and 3 out of 6 executors might send succession messages (containing secrets of which 5 are needed to recover the relay's private key). We need to record the fact that if two more are transmitted, the successor will have formally inherited the nonresponder's duties. We also need to know which 3 have been sent, so we don't double-count.
- Which relays have been disqualified for not responding or bad behaviour such as sending bad secrets or complaining about valid ones. At the end of the diurn, relays that haven't been disqualified get a fee.

This is what a starting user needs to know (and what everybody must agree on) in order to determine which relays will be selected to hold which secrets during trades. The choice of relays is deterministically chosen based on the relay state and other information.

The hashes of the spent chain and the relay state are encoded in the mined credit, along with information which tracks the relationship of this mined credit to the diurn, namely:

- The diurnal block root. As mined credits appear in succession, their hashes get added to a list of hashes called the diurnal block. These are then taken to be the leaves of a hash tree whose root is the diurnal block root.

  This means that, once this diurn is over, for the purposes of being able to validate any CreditInBatch objects which are inputs to transactions, we can forget about all the mined credits (containing batch roots), and we just need to remember the diurnal block root.

  The second branch in the CreditInBatch object, the diurnal branch, $g_1, g_2, \cdots, g_n$, connects the hash of the MinedCredit to the diurnal block root. So there's a chain of proof $h_1, h_2, \cdots, h_m$, from the Credit to the CreditBatch and from the hash of the MinedCredit containing that CreditBatch to the diurnal block root.

  The MinedCredit hash is constructed in a special way: The hash of everything except the batch root is calculated first, and called the branch bridge, r. Then the same symmetric hash function used for constructing trees and branches is used to combine the branch bridge with the batch root. This means that there's effectively a long branch, $g_1, g_2, \cdots, g_n, r, h_1, h_2, \cdots, h_m$, connecting each valid credit all the way to the diurnal block root stored in the calendar.

  At the end of the diurn, when the final diurnal block root becomes known, everybody who received a payment during the previous diurn should construct the diurn branches connecting their CreditInBatch objects to the diurnal block root. Later, anybody who has a calendar can validate those CreditInBatch objects, because every diurnal block root is stored in the calendar.

- The previous diurn root. Apart from the diurnal block root (the root of an ordered collection of hashes), each diurn has a "diurn root", which is obtained by combining (and hashing) the diurnal block root with the previous diurn's root, or 0 if there was no preceding diurn.

  This allows us to prove that each calend (the mined credits which are stored in the calendar and indicate the end of one diurn and the start of the next) follows the previous calend.

- The diurnal difficulty. This is a number which is adjusted after each calend. If the previous diurn was less than 24 hours in length, it is increased. If it was longer, it is decreased. A mined credit whose proof of work produces a hash smaller than 2 ** 128 divided by the diurnal difficulty, is a calend.

  To take into account the possibility that the hashrate could drop, small decrements are made to the diurnal difficulty every time a new mined credit appears.

When mined credits are sent, they are enclosed in a MinedCreditMessage:

MinedCreditMessage		
Field Name	**Data Type**	**size in bytes**
mined_credit	MinedCredit	238
proof	TwistWorkProof	172 + 12 * L
hash_list	ShortHashList	20 + 4 * N
timestamp	uint64_t	8

The proof field is the proof of work which has to exceed the difficulty threshold in order for the mind credit to be accepted. The number L is the number of "links" in the proof and is about 512 on average. I have developed a later version of the proof of work which can reduce the proof size to about 210 + 3 * L, but this version is simpler and will work for now. More details about the proof of work will follow.

The hash_list object contains 4 bytes of the 20-byte hashes of every message (transactions, mined credit messages, relay messages and so on) since the previous mined credit. It also contains a 20-byte disambiguator consisting of the xor of all the 20-byte hashes, which can be used to handle collisions among the possible 4-bytes hashes. The hash of this list itself is contained in the mined credit, so that an attacker can't alter it without invalidating the message.

If the full 20-byte hashes can't be recovered from the hash_list, it means that some of the messages referred to have not yet been received, in which case the recipient of the MinedCreditMessage will ask the sender for the missing messages.

The calendar looks like:

Calendar		
Field Name	**Data Type**	**size in bytes**
calends	std::vector<MinedCreditMessage>	variable
current_diurn	Diurn	variable
extra_work	vector<vector<pair<MinedCredit, TwistWorkProof>>>	variable
current_extra_work	vector<pair<MinedCredit, TwistWorkProof>>	variable

Because the appearance of calends is stochastic, it can't be absolutely guaranteed that the sum of the diurnal difficulties will always equal or exceed the sum of the difficulties of each mined credit.

Sometimes, the sum of the batch difficulties, which is recorded in the total_work field of each mined credit, will exceed the sum of the diurnal difficulties of the calends plus the work in the current diurn. Whenever a mined credit appears for which this is the case, we store that mined credit and its proof of work as "top up work" to ensure that the total amount of work in the calendar (the diurnal work in the calends plus the batch work in the current diurn plus the top up (batch) work) always equals or exceeds the total_work reported in the mined credit[1].

[1] An analogy can be made to diamond mining. Suppose that you can only obtain diamonds in the following wasteful way: You specify the size of the diamond you want, and then a robot digs and digs until it finds a diamond greater than or equal to your requested size. Then it cuts the diamond down to exactly the size you requested, and wastefully destroys the rest. Then it repeats the process, bringing you many diamonds over time.

Cryptocurrency mining (i.e. looking through hashes for a small one) is a bit like this. If you're looking for a small hash and you find a tiny one, it only proves that you successfully found a small one. You didn't successfully find a tiny hash because you never tried to do that. The same hash, for a person seeking a tiny hash, proves a lot more work. It's what you were looking for (the difficulty), not what you find (the hash), that measures how much work

Sometimes diurns will be shorter than average, and on these occasions, the diurnal work will exceed the total batch work. When this occurs, some of the top up work (from the oldest diurns) can be discarded, shrinking the calendar.

A diurn consists of:

Diurn		
Field Name	**Data Type**	**size in bytes**
previous_diurn_root	uint160	20
diurnal_block	vector<uint160 >	variable
credits_in_diurn	vector<MinedCreditMessage >	variable
initial_difficulty	uint160	20
current_difficulty	uint160	20

Similar to a CreditBatch, a Diurn object generates branches connecting each mined credit (and its enclosed batch of credits) to the diurn root. Since the diurn root can be obtained from the calends which are stored, it's not necessary, for the purposes of validating transaction inputs, to store the contents of the diurn after it's over, or to download it when initializing.

Of the data structures described above, only MinedCreditMessages, SignedTransactions and Calendars are transmitted.

5 Mining

The miner constructs a serialized MinedCredit object, takes its hash, and begins to construct a proof of work using the 256-bit hash as the input.

5.1 The Twist Proof of Work

5.1.1 Simplified Version

The work performed consists of filling 4 Gb of memory with pseudo-random data and then pseudo-randomly generating a sequence of numbers, interpreting each new number as pointing to a specific location within that 4 Gb, which (being binary data) is also interpreted as a number. The number generated in the sequence is combined with the number to which it points and the result is hashed. If the hash is sufficiently small, the number of steps taken is recorded and the work is over.

The worker can present the report: "I filled up 4 gigabytes and then found a hash below the threshold after exactly two million and three iterations."

If we're checking this proof of work thoroughly, we'll allocate 4 Gb and not only check that a hash sufficiently small is encountered (at the relevant location in the 4 Gb) after two million and three hashes, but we'll also check that no sufficiently small hash is encountered before that iteration.

If we find a hash smaller than the threshold at an earlier iteration, at a specific place in memory, then it means that the worker presenting this proof didn't look at that part of memory, because if he had, he would have stopped and published the proof. So he was cheating, filling up less than 4 Gb and hoping that the location in memory where the true sequence stops is inside the area of memory he has allocated.

This can be characterized as a proof of work scheme, but it has several undesirable features. One is that the proof checker must expend as much time and memory to check the proof as the worker does to create it. If it took a minute of mining to create it, it'll take a minute to check it.

Another undesirable feature is that the cheater described above has a reasonable chance of finding a valid proof with only 2 Gb or even 1 Gb of ram.

you did.

The calends are analogous to big diamonds which were sought and successfully found, and the regular mined credits are like smaller, more frequent diamonds.

5.1.2 Less Simple Version

We use the old two-threshold trick again to mitigate the two problems described above. There's a threshold for ending the proof, and a more lax threshold. Hashes low enough to pass the lax threshold occur on average N times more frequently than hashes low enough to pass the threshold for ending the proof[2].

To distinguish the two thresholds, the threshold for ending the proof will be referred to as the target, while the laxer threshold will be called the link threshold.

When the link threshold is passed:

- The random number generator is reseeded with the number at the specified location in memory. That number is then called a link and recorded as part of the proof.
- The number of steps since the last link (or the start if there was no previous link) is recorded as well.

When the target is finally reached, a list of pairs of numbers has been recorded. This can be presented, along with the original seed which was used to fill the memory and seed the random number generator, as a proof of work.

To thoroughly check the proof of work, it will, once again, be necessary to allocate as much memory and work for as long as the worker did to produce the proof. However, we can do a quick check by starting at the last link. The link is the seed of the RNG, so we can seed our RNG with it and start iterating. If the proof is valid, we'll reach the target in approximately one N^{th} of the time it took the prover to reach it.

Of course this quick check doesn't prove that the worker allocated 4 Gb of ram for the entire duration of the work. The worker might have allocated only 1 kilobyte of ram and then iterated and iterated until the target was reached within that 1 kilobyte. But it does prove that the work was done (a hash below the target was successfully found), and that at the very least, the part of memory where the target was reached was allocated.

A checker can do spot checks: he or she can choose a few links at random and check that the procedure of iterating the random number generator, seeded with the link, combining the generated number with the number it points to in the 4 Gb, and hashing the result, reaches the link threshold after exactly the specified number of iterations and not before, and that the next link reached is indeed the one listed in the proof.

TwistWorkProof		
Field Name	**Data Type**	**size in bytes**
memory_seed	uint256	32
target	uint160	20
link_threshold	uint160	20
quick_verifier	uint160	20
num_links	uint32_t	4
num_segments	uint32_t	4
seeds	std::vector<uint8_t>	1
links	std::vector<uint64_t>	8 * L
linklengths	std::vector<uint32_t>	4 * L

where:

- **memory_seed** - The hash of the MinedCredit object prepared by the miner. This is used to seed a random number generator which fills up all 4 Gb of memory. It also deterministically determines the seed for the random number generator which iterates from link to link.

[2] N is currently set to 512 in the Flex mining proof of work.

- **target** - Each time a number, X, is generated, $X \bmod N$ is calculated where N is the number of numbers that fit into 4 Gb. The number located at position $X \bmod N$ is then xored with X and the result is hashed to achieve this iteration's hash. If the hash is less than the target, the number of iterations since the last link is recorded and the proof has been completed.

  If the hash is greater than both the target and the link_threshold, the next random number is generated and the next iteration begins.

- **link_threshold** - This number is larger than the target, and hashes fall below this threshold much more frequently. When that occurs, the number located at $X \bmod N$ in memory is used to seed the random number generator for the next iteration. This number is appended to the links vector in the TwistWorkProof, and the number of iterations since the last link is appended to the linklengths vector.

- **quick_verifier** - This contains the last number generated in the proof. By combining it with the number to which it points in memory and hashing the result, a number below the target should be achieved. This provides a quick way of checking that the work was done, although it doesn't prove that the work was done in series or that all 4 Gb of RAM was occupied.

- **num_segments** - The scheme used to fill up the 4 Gb of memory does it in a modular way. The 4 Gb is filled up in independently-generatable segments. There are 64 segments in the proof of work used for mining Flex credits.

- **seeds** - Each segment is generated by combining one byte (a seed), with the segment number (2 for the second segment, 3 for the third etc) and also with a number deterministically generated from the memory_seed. The first seed is obtained from the memory_seed. The second seed is obtained from bytes at the end of the first segment. The third seed is taken from the bytes at the end of the second segment and so on.

  These are stored so that proof checkers can generate the memory segments in parallel. Each proof checker can choose a few segments at random, choose a few links at random, and iterate from one link to the next, constructing hashes on those occasions when $X \bmod N$ falls within one of the segments they have constructed.

- **links** - These are 64-bit numbers which are the bytes in memory at the location $X \bmod N$ on the occasions when X xored with those bytes[3] and hashed and the result is less than the link threshold.

  Each link is used as the seed of the random number generator for the next iteration, so a checker can start checking the proof at any link, and needn't check the entire proof from start to end.

- **linklengths** - Each link length is the number of iterations needed to reach the next link. Whenever a link is encountered, the number of iterations since the last link is recorded by appending it to the linklengths vector.

Each checker can check that no links are encountered, other than the ones specified in the proof of work, within the segments of memory that the checker decides to check. With mined credits in Flex, the burden of checking the proof of work is shared across the network.

Each recipient of a proof of work checks the quick_verifier, X, to ensure that the target is reached. This requires generating only 1 segment, namely the one containing the part of memory pointed to by X.

[3]In the actual implementation, 8 bytes of X are xored with the first 8 of 16 bytes at those locations, and the result is padded out to the required length and hashed. This is to reduce the already low probability of getting into a cycle during the work.

If it takes 6 seconds to fill up 4 Gb of ram in 64 segments, and one minute to complete a proof of work with 512 links, then it takes about $\frac{6}{64} + \frac{60}{512} \approx 0.2$ seconds to do a quick check of any link, while it takes less than 0.1 seconds to check the quick_verifier.

The proofs of work are quite large, principally because of the number of links, each of which contributes 12 bytes to the size of the message. This could be reduced by storing (say) only every 4th link and storing only the number of iterations between each 4th link (i.e. from 0 to 4, from 4 to 8 etc) along with three bytes, where the bytes specify which memory segments the other 3 links (the 3 that weren't stored) will be encountered in. This makes it possible for a checker to discover for himself or herself 3 out of every 4 links checked, and to confirm that they are located in the specified segments.

5.1.3 Pseudo-Code

Filling Memory To fill memory, the following algorithm is used:

```
Given:
------
X_in:   a 256-bit input
N:      Memory usage parameter
r:      block size parameter
S:      Desired number of independently-generatable memory segments


Output:
-------
V:      N blocks of data, composed of S segments,
        with each segment containing N / S blocks
Seeds:  S bytes, where the ith byte, once known, can be used with
        X_in and i to generate the ith segment

Sizes
------
Segment = N / S blocks
Block = 512 Words
Word = 8 bytes = 64 bits


Algorithm
---------
1:  Let NumSeeds = 0
2:  Let B = a block of 512 words all set to zero
3:  Compute X = 512 word hash generated by applying Blake512 to X_in
4:  Set first byte of B = first byte of X
5:  for i = 0 to N:
6:      if i % SegmentSize == 0:
7:          V[i] = PrepareSegmentStartBlock(X, N, r, S, i, B)
9:      else:
10:         V[i] = PrepareBlock(X, N, r, S, i, V, B)
11:     B = V[i]
12: end for
```

Where PrepareSegmentStartBlock(X, N, r, S, i, B) is defined by:

```
Seeds[NumSeeds] = first byte of B
for j = 0 to 512:
    B[j] = X[j] * (1 + NumSeeds) - j
end for
first byte of B = Seeds[NumSeeds]
NumSeeds += 1
return BlockMix(B, r)
```

and PrepareBlock(X, N, r, S, i, V, B) is defined by:

```
SegmentLocation = i % SegmentSize
SegmentStart = i - SegmentLocation
for k = 0 to 512:
    mixer = B[k] % (SegmentLocation * 512)
    B[k] = B[k] xor V[SegmentStart * 512 + mixer]
end for
return BlockMix(B, r)
```

where BlockMix is the function defined in Colin Percival's Scrypt paper[4] as:

```
Algorithm BlockMix$_{H,r}(B)$

Parameters:
H A hash function.
r Block size parameter
Input:
$B_0, \cdots, B_{2r-1}$ Input vector of 2r k-bit blocks

Output:
$B_0, \cdots, B_{2r-1}$

Output vector of 2r k-bit blocks.

Steps:

$X \leftarrow B_{2r-1}$
for i = 0 to $2r - 1$ do:
    $X \leftarrow H(X$ xor $B_i)$
    $Y_i \leftarrow X$
end for
$B \leftarrow (Y_0, Y_2, \ldots Y_{2r-2}, Y_1, Y_3, \cdots, Y_{2r-1})$
```

and H is Salsa64/20.

Sequentially Searching for the Target In the code below, the byte 85 (01010101) is used to pad inputs to the Salsa64/20 hash function up to 512 words. The notation X[1:512] = 85 means set all of the bytes between X[1] and X[512], including X[1] but not X[512] to 85.

[4]Stronger key derivation via sequential memory-hard functions, https://www.tarsnap.com/scrypt/scrypt.pdf

The search algorithm is:

```
Given:
X_in: a 256-bit input
V:    N * 512 words of data
L:    A link threshold
T:    A target

1:  Let Y = 512 words, all set to zero
2:  X[0] = X_in[0]
3:  X[1:512] = 85
4:  c = 0
5:  while true:
6:      X = Salsa64(X)
7:      c = c + 1
8:      j = X[1] % (N * 512)
9:      j = j - (j % 2)
10:     Y[0] = X[0] ^ V[j]
11:     Y[1] = V[j + 1]
12:     Y[2:512] = 85
13:     Y = Salsa64(Y)
14:     Z = 128 bit number taken from the first 16 bytes of Y
15:     if Z < L:
16:         Append V[j] to Links
17:         Append c to LinkLengths
18:         c = 0
19:         X[0] = V[j]
20:         X[1:512] = 85
21:     end if
22:     if Z < T:
23:         break
24:     end if
25: end while
```

6 Cryptocurrency Trading

All trades on the exchange are between the exchange's cryptocurrency, FLX, and a foreign currency. FLX is the numéraire, so in each trade, there's a buyer, who pays FLX in exchange for another currency, and a seller, who receives FLX in exchange for the other currency. Each order placed on the exchange is either a buy order for a currency or a sell order.

To simplify the terminology, specific amounts of the foreign currency will be referred to as currency or coins, while specific amounts of FLX will be referred to as credits. Each trader buys or sells currency for credits.

There are four main steps involved in a trade before any FLX or other currency is sent anywhere:

- An order is placed
- The order is accepted
- The trader who placed the order commits to the trade

- The accepter commits to the trade

At each stage, cryptographic information is transmitted, so that when the four steps are complete, the traders know what cryptocurrency addresses they should send their payments to. By then, the relays collectively have enough information to give either trader the private keys to either address.

The Order message contains:

Order		
Field Name	**Data Type**	**size in bytes**
currency	vch_t	variable
side	uint8_t	1
size	uint64_t	8
price_per_million	uint64_t	8
is_cryptocurrency	uint8_t	1
order_pubkey	Point	34
fund_address_pubkey	Point	34
fund_proof	vch_t	variable
relay_chooser_hash	uint160	20
proof_of_work	TwistWorkProof	variable
timestamp	uint64_t	8
timeout	uint32_t	4
pubkey_authorization	Signature	64
confirmation_key	Point	32
auxiliary_data	vch_t	variable
signature	Signature	64

The fields are:

- **currency** - This is the currency code such as BTC, NMC and so on.
- **side** - This is either 0 (bid or buy) or 1 (ask or sell).
- **size** - This is the number of satoshis (smallest unit) of the currency that the trader wishes to buy or sell.
- **price_per_million** - This is the integer number of smallest units of FLX (commonly called satoshi) that should be exchanged for 1,000,000 of the smallest units of the other currency.
- **is_cryptocurrency** - This is 1 if the currency is a cryptocurrency, 0 otherwise.
- **order_pubkey** - This is the public key that the trader will use to communicate (signatures and encryption) during this trade. This is always a secp256k1 point, and if the currency uses the curve secp256k1, this must be the same as the fund_address_pubkey.
- **fund_address_pubkey** - This is the public key of a cryptocurrency address which holds the funds for the trade.
- **fund_proof** - This a string of bytes which can convince the other trader that the funds for the trade are in fact at the specified address. For FLX credits, this is a serialized CreditInBatch object. For bitcoin-like currencies, it's a transaction id plus a number specifying which UTXO of the transaction pays to the address. It could be expanded to included multiple UTXO's.
- **relay_chooser_hash** - The traders will need to select relays to hold parts of secrets. They need to be confident that the other trader doesn't have any influence over which relays are selected. To achieve this, the trader placing the order picks a number, called the relay_chooser, and publishes its hash.

Whoever accepts the order will publish a relay_chooser. Then the order placer (when committing to the trade) will reveal the relay_chooser whose hash is the relay_chooser_hash. The two relay choosers, once known, are combined to deterministically choose the relays.

Because each trader committed irreversibly to the choice of relay_chooser before discovering the other trader's choice, each one knows that the final deterministic choice of relays was unknowable to the other trader.

- **proof_of_work** - The traders can include an optional proof of work, with a difficulty of their choosing, with orders and accept order messages. If there is too much traffic on the network, messages with more difficult proofs of work will be transmitted across the network with a higher priority.
- **timestamp** - The time when the order was generated, in microseconds.
- **timeout** - How long the relays should wait for the trade to complete before sending the credits (if they've been paid) back to the buyer and the coins (if they've been paid) back to the seller.
- **pubkey_authorization** - The currency seller must prove that the owner of the coins is the person communicating via the order_pubkey, which is a secp256k1 point. This is accomplished by signing the order_pubkey with the private key that only the fundholder knows.

  This is unnecessary for secp256k1 cryptocurrencies, for which the fund_address_pubkey can be used to communicate.
- **confirmation_key** - This is the key of a third party authorized to notarize fiat payments. It's not used for cryptocurrencies.
- **auxiliary_data** - This is data provided by the trader that the third party may need in order to verify the occurrence of a fiat payment. E.g. An account number or email address, possibly encrypted so that only the third party can read it.
- **signature** - The order, and subsequent messages from this trader, must be signed using the order_pubkey.

  When creating signatures for messages, the signature field is first set to zero. Then the message is hashed, and this field is populated with the signature of the hash.

The AcceptOrder message contains:

AcceptOrder		
Field Name	**Data Type**	**size in bytes**
currency	vch_t	variable
order_hash	uint160	20
accept_side	uint8_t	1
order_pubkey	Point	34
fund_address_pubkey	Point	34
fund_proof	vch_t	variable
relay_chooser	uint160	20
previous_credit_hash	uint160	20
proof_of_work	TwistWorkProof	variable
pubkey_authorization	Signature	64
confirmation_key	Point	34
auxiliary_data	vch_t	variable
signature	Signature	64

The fields are:

- **currency** - This is the currency code such as BTC, NMC and so on.
- **order_hash** - Hash of the order being accepted.
- **accept_side** - The opposite of the order side (=1-order side).
- **order_pubkey** - The accepter's secp256k1 public key for communicating.
- **fund_address_pubkey**
- **fund_proof**
- **relay_chooser**
- **previous_credit_hash** - This is the hash a mined credit which has appeared recently. The relays will be chosen from the RelayState whose hash is encoded in the specified mined credit.
- **proof_of_work**
- **pubkey_authorization**
- **auxiliary_data**
- **signature**

If the order placer receives a valid AcceptOrder message, he or she will then send an OrderCommit message:

OrderCommit		
Field Name	**Data Type**	**size in bytes**
curve	vch_t	variable
order_hash	uint160	20
accept_order_hash	uint160	20
relay_chooser	uint160	20
second_relay_chooser_hash	uint160	20
distributed_trade_secret	DistributedTradeSecret	variable
shared_credit_secret_xor_shared_secret	CBigNum	32
shared_credit_pubkey	Point	34
shared_currency_secret_xor_hash_of_shared_credit_secret	CBigNum	32
shared_currency_pubkey	Point	34
signature	Signature	64

The fields are:

- **curve** - A byte indicating the elliptic curve.
- **order_hash** - Hash of the order being accepted.
- **accept_order_hash** - Hash of the AcceptOrder message.
- **relay_chooser**
- **second_relay_chooser_hash** - The secrets will need to be disclosed to a second set of relays. The same technique is used to make them unpredictable.

- **distributed_trade_secret** - This contains two sets of 34 secrets for 34 relays.

  The scheme is: private key = $a + b + c$, where c is a shared secret (shared_credit_secret or shared_currency_secret), a is a number chosen by one trader, and b is a number chosen by the other trader.

  The numbers a and b are split into 34 parts, any 31 of which can be combined to reproduce a or b. Each trader chooses two such numbers (one for credits and one for the currency).

- **shared_credit_secret_xor_shared_secret** - There's a need for the traders to share a secret that the relays don't know (c above), so that the relays can't run away with any money.

  To share the secret, a Diffie-Hellman style shared secret ("shared_secret") is generated, and this is xored with the secret ("shared_credit_secret") that one trader wishes to send to the other.

- **shared_credit_pubkey** - This is the secp256k1 elliptic curve point, C, corresponding to the shared credit secret. The public key $A + B + C$ is public, so everybody gets to see A, B and C.

- **shared_currency_secret_xor_hash_of_shared_credit_secret** - This is similar to the previous field, but intended to share the corresponding secret which will be used to construct the address to which the coins will be sent. If we use d, e and f to correspond to the currency's versions of a, b and c, then the shared currency secret is f, while d and e are each shared between the relays (via the 31-of-34 scheme) and one trader.

- **shared_currency_pubkey** - The elliptic curve point (possibly Ed25519 or another curve) corresponding to the shared_currency_secret. A similar address constructed of points D, E and F is created for the cryptocurrency being traded, but these points have to be on the currency's curve, which might not be secp256k1.

- **signature**

Presuming the order accepter validates all the information in the OrderCommit, he or she will send an AcceptCommit message:

AcceptCommit		
Field Name	**Data Type**	**size in bytes**
curve	vch_t	variable
order_hash	uint160	20
order_commit_hash	uint160	20
second_relay_chooser	uint160	20
distributed_trade_secret	DistributedTradeSecret	variable
signature	Signature	64

The fields are:

- **order_hash**
- **order_commit_hash** - Hash of the OrderCommit.
- **second_relay_chooser**
- **distributed_trade_secret**
- **Signature**

Having exchanged these messages, both traders know A, B, C, D, E and F. Credits are sent to $A + B + C$; coins are sent to $D + E + F$.

Both traders know c and f. One trader knows a and d. The other trader knows b and e.

The relays, if 31 of them were to conspire, could reproduce a, b, d or e. When operating correctly, the relays can provide any of these numbers to one of the traders in the event that the other trader doesn't courteously send it along.

The result is that both traders can have confidence that either the trade will go through, or they will at least get their money back.

The DistributedTradeSecret object contains:

DistributedTradeSecret		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
relay_chooser	uint160	20
relay_list_hash	uint160	20
credit_coefficient_points	std::vector<Point>	variable
currency_coefficient_points	std::vector<Point>	variable
pieces	std::vector<PieceOfSecret>	variable

where the fields are:

- **credit_hash** - The hash of a recent mined credit whose RelayState the relays are to be chosen from.
- **relay_chooser** - A seed for a random number generator which will choose the relays.
- **relay_list_hash** - The hash of the list of chosen relays. Including this makes it clear that everybody agrees on what relays were chosen.
- **credit_coefficient_points** - The K-of-N secret sharing scheme works as follows. The trader generates a random $(K - 1)^{\text{th}}$ order polynomial, P, with coefficients in $\mathbb{Z}$ mod M, where M is the modulus of the curve group.

  The secret is the value of $P(0)$, which is equal to the 0^{th} coefficient: $P(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{K-1}x^{K-1}$.

  The secrets for the N relays are the values $P(1), P(2), \cdots, P(N)$. These are communicated in the pieces field.

  The credit_coefficient_points are the elliptic curve points $C_0, C_1, \cdots$, corresponding to the coefficients $c_0, c_1, \cdots$.

  Each relay can use these coefficients to verify that the secret he or she received is correct. E.g. If the secret sent to the 2^{nd} relay is y, then the relay checks that $yG = Y = C_0{+}2C_1{+}2^2C_2{+}\cdots$, where G is the generator of the curve group.

  Somebody who knows two points on a line can reconstuct the equation of the line. For a quadratic curve, three points are needed. For a cubic, four are needed and so on. This is done using Lagrange's interpolating polynomials.

  So somebody who receives K of the N values of the $(K - 1)^{\text{th}}$ order polynomial, P, can reconstruct the equation of P, and then calculate the value of $P(0) = c_0$.
- **currency_coefficient_points** - These are similar to the credit_coefficient_points, but they're used for the currency instead of credits, and the elliptic curve might not be secp256k1.
- **pieces** - These are the individual secret parts sent to the relays.

The PieceOfSecret object looks like:

PieceOfSecret		
Field Name	**Data Type**	**size in bytes**
credit_pubkey	Point	34
currency_pubkey	Point	34
credit_secret_xor_shared_secret	CBigNum	32
currency_secret_xor_hash_of_credit_secret	CBigNum	32

where the fields are:

- **credit_pubkey -** There are two secrets that each relay receives from each trader: a credit_secret (e.g. part of a) and a currency_secret (e.g. part of d). credit_pubkey is the elliptic curve point corresponding to the credit_secret.

  The relay, having recovered the value of the secret part, multiplies the secret by G, the group generator, and compares the result to the credit_pubkey. If they differ, the secret is invalid and the relay will complain.

- **currency_pubkey -** Same as credit_pubkey, but for the currency.

- **credit_secret_xor_shared_secret -** A shared_secret is used to communicate the credit_secret.

  To generate the shared_secret, the trader multiplies the credit_secret by the relay's public key and then hashes the serialization of the resulting Point. The relay generates the same shared_secret by multiplying his or her private key by the credit_pubkey.

  Having done this, the relay xor's the shared_secret with the credit_secret_xor_shared_secret to recover the credit_secret. The relay then checks that multiplying the credit_secret by the group generator gives the credit_pubkey, and complains if it doesn't.

- **currency_secret_xor_hash_of_credit_secret -** This is simlar to the previous field, but communicates the currency_secret.

After the four main messages have been sent, the order placer knows both the order accepter's second_relay_chooser as well as his or her own second_relay_chooser, whose hash was published in the OrderCommit message.

The order placer combines these to choose another set of 34 relays and sends a DistributedTradeSecretDisclosure message:

DistributedTradeSecretDisclosure		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
relay_chooser	uint160	20
relay_list_hash	uint160	20
revelations	std::vector<RevelationOfPieceOfSecret >	variable

with fields:

- **credit_hash -** To specify the RelayState from which the relays should be chosen.

- **relay_chooser -** Xor of the second_relay_choosers chosen by the two traders. This is used to seed the RNG to choose the relays.

- **relay_list_hash -** So we definitely agree on which relays were chosen.

- **revelations -** These reveal the secrets to 34 newly chosen relays.

Each RevelationOfPieceOfSecret contains:

RevelationOfPieceOfSecret		
Field Name	**Data Type**	**size in bytes**
credit_secret_xor_second_shared_secret	CBigNum	32

where:

- **credit_secret_xor_second_shared_secret -** The second_shared_secret is generated by the trader by muliplying the credit_secret by the public key of the relay to whom the secret (or rather, part of the secret) is being revealed. The serialization of the resulting Point is then hashed.

  The relay uses information which is sent alongside the DistributedTradeSecretDisclosure to retrieve the original credit_pubkey corresponding to this credit_secret. The relay multiplies his or her private key by the credit_pubkey to generate the second_shared_secret.

  The relay xor's the second_shared_secret with the credit_secret_xor_second_shared_secret to recover the credit_secret.

This reveals the individual credit_secret to the second relay who gets sent that secret and can validate it by multiplying it by the group generator and comparing the result to the original credit_pubkey sent in the original DistributedTradeSecret.

This relay also has the duty of checking if the first relay to be sent this secret received the same one.

To do this, the relay retrieves the original DistributedTradeSecret and finds the relevant credit_secret_xor_shared_secret. He or she then tries to reconstruct the shared_secret, which was shared between the trader and the earlier relay.

This is done by taking the credit_secret which was recovered using the RevelationOfPieceOfSecret, and multiplying that credit_secret by the earlier relay's public key. The resulting Point is then serialized and hashed. This is in fact how the trader originally constructed the shared_secret.

Finally, the relay doing the checking xor's the reconstructed shared_secret with the credit_secret_xor_shared_secret and compares the result to the credit_secret he or she was sent. If they match, it means the original relay received the same secret.

A similar check is performed for the currency_secret, but the trader doesn't need to send any additional data for that. The hash of the credit secret is xored with the currency_secret_xor_hash_of_credit_secret field from the original DistributedTradeSecret. This makes the check possible without increasing the size of the already large messages.

If either secret doesn't match what was sent to the previous trader, or doesn't match its pubkey, then it means that the trader is sending bad data, so the relay will complain and the other trader won't go through with the trade.

Presuming that all four messages have been sent and there haven't been any complaints (both traders will wait for a while to listen for complaints before proceeding), the traders will send their coins and credits to the appropriate addresses.

The fact that the buyer has sent credits is verified within the network by the relays and by the seller. The relays have no direct way to confirm that the currency has been sent to the address $D + E + F$ by the seller, so they wait for the buyer to inform them that this has occurred.

If the traders are using the reference client, the following will happen automatically:

The buyer will send the credits to $A + B + C$ and the seller will send the currency to $D + E + F$ and then send a CurrencyPaymentProof:

CurrencyPaymentProof		
Field Name	**Data Type**	**size in bytes**
accept_commit_hash	uint160	20
proof	vch_t	variable
signature	Signature	64

with fields:

- **accept_commit_hash -** This identifies the AcceptCommit, and via it, the other messages relevant for the trade.
- **proof -** This is a txid + output number combination for bitcoin-like cryptocurrencies.
- **signature**

The buyer, upon receiving and validating this, will wait for a specified number of confirmations and then send a CurrencyPaymentConfirmation:

CurrencyPaymentConfirmation		
Field Name	**Data Type**	**size in bytes**
order_hash	uint160	20
accept_commit_hash	uint160	20
payment_hash	uint160	20
affirmation	uint8_t	1
signature	Signature	64

with fields:

- **order_hash**
- **accept_commit_hash**
- **payment_hash -** This is the hash of the CurrencyPaymentProof.
- **affirmation -** This is a byte set to a specific value just to represent the buyer's formal acknowledgement of the payment.
- **signature**

Each trader, having confirmed that the funds have reached the addresses, $A{+}B{+}C$ and $D{+}E{+}F$, will send a CounterPartySecretMessage:

CounterPartySecretMessage		
Field Name	**Data Type**	**size in bytes**
accept_commit_hash	uint160	20
sender_side	uint8_t	1
secret_xor_shared_secret	CBigNum	32
signature	Signature	64

with fields:

- **accept_commit_hash**
- **sender_side -** 0 for buyer, 1 for seller.
- **secret_xor_shared_secret -** The same scheme as is used for sending a credit_secret is used for sending this secret, which is either a currency secret such as d (if sent by the seller) or a credit secret such as b (if sent by the buyer).

- **signature**

If one of the traders complains about the CounterPartySecretMessage, or if the other trader never sends one, each of the first set of 34 relays will send a TradeSecretMessage:

TradeSecretMessage		
Field Name	**Data Type**	**size in bytes**
accept_commit_hash	uint160	20
direction	uint8_t	1
position	uint32_t	4
credit_secret_xor_shared_secret	CBigNum	32
currency_secret_xor_shared_secret	CBigNum	32
signature	Signature	64

with fields:

- **accept_commit_hash**
- **direction** - 0 for forwards (trade is to be completed), 1 for backwards (trade is to be cancelled).

  If the direction is forwards, the credit_secret is for the seller and the currency_secret is for the buyer. If it's backwards, it's the other way around.
- **credit_secret_xor_shared_secret**
- **currency_secret_xor_shared_secret**
- **signature**

Each trader needs 31 such messages to reconstruct the private key needed to spend the coins or credits. The second set of 34 relays can quickly send any secrets needed in case more than 3 of the first set are not online. The duties of those who have gone offline will be inherited by their successors, who can then provide any secrets for which both the relays in the first and the second set have gone offline.

7 Relays

When a credit is mined, the miner can apply to become a relay. To do this, the miner sends a RelayJoinMessage:

RelayJoinMesssage		
Field Name	**Data Type**	**size in bytes**
relay_number	uint32_t	4
credit_hash	uint160	20
preceding_relay_join_hash	uint160	20
relay_pubkey	Point	34
successor_secret_xor_shared_secret	CBigNum	32
successor_secret_point	Point	34
distributed_succession_secret	DistributedSuccessionSecret	644
signature	Signature	64

where:

- **relay_number** - The first relay has relay number 1. The second has relay number 2, and so on.

- **credit_hash** - This is the hash of the credit that this aspiring relay has mined.
- **preceding_relay_join_hash** - Relays must be assigned a number which everybody agrees on. To achieve this, each relay specifies which relay was the last one to apply before this one. This is done by including the hash of the previous relay's RelayJoinMessage.
- **relay_pubkey** - This is the secp256k1 public key that the relay will use for communicating.

  This is distinct from the key which is specified in the keydata field of the mined credit, namely the key which can spend the credit. The reason to use a different key for relay communication is that the private key corresponding to the relay_pubkey might be reconstructed by a successor relay with the assistance of executors. If that happens, the successor should not get the ability to spend the mined credit.
- **successor_secret_xor_shared_secret** - The relay's private key is expressed as the sum of two secrets: a successor_secret and a succession_secret.

  The successor_secret is sent to the relay's successor and the succession_secret is broken into 6 parts and sent to six executors.

  To communicate the successor_secret to the successor, the relay constructs a shared_secret by multiplying the successor_secret by the successor's relay_pubkey. The resulting Point is then serialized and hashed.

  The successor_secret_xor_shared_secret is then calculated by xoring the successor_ secret with the shared_secret.
- **successor_secret_point** - The successor_secret_point is the elliptic curve point corresponding to the successor_secret. The joining relay calculates it by multiplying the successor_secret by the generator of the curve group.

  The successor can generate the shared_secret by multiplying his or her private key by the successor_secret_point and then serializing and hashing the resulting Point. The shared_secret can then be xored with the successor_secret_xor_shared_secret to recover the successor_secret.

  Once the successor_secret has been recovered, the successor can check that it's valid by multiplying it by the group's generator. If the result is equal to the successor_secret_point, then the secret is valid. Otherwise, it's a bad secret and the successor will complain.
- **distributed_succession_secret** - The joining relay's private key is expressed as a successor_secret plus a succession_secret.

  Whoever knows both of these secrets can reconstruct the relay's private keys and decrypt all the secrets that have been sent to that relay.

  The successor is sent the successor_secret, and the succession_secret is broken into 6 pieces, any 5 of which can be combined to reconstruct the succession_secret, and these six pieces are sent to six relays called executors.

  In the event that this relay stops responding while holding secrets needed to complete a trade, any of the 6 executors who are still online and functioning correctly will send their parts of the succession_secret to the successor.
- **signature** - The message must be signed using the key which can spend the mined credit. This proves that the message was created by miner.

  Further communications from this relay must be signed with the private key corresponding to the relay_pubkey.

The DistributedSuccessionSecret looks like:

DistributedSuccessionSecret		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
relay_number	uint32_t	4
relay_list_hash	uint160	20
coefficient_points	std::vector<Point>	6×34
point_values	std::vector<Point>	6×34
secrets_xor_shared_secrets	CBigNum	6×32

where:

- **credit_hash** - This and the relay_number are used to used to specify which relay is applying to join and the state of the network at that time (encoded in the specified mined_credit).
- **relay_number**
- **relay_list_hash** - This is the hash of the list of executors.
- **coefficient_points** - The 5-of-6 secret sharing scheme works as follows:

  The succession_secret is denoted c_0, and 4 random numbers smaller than the modulus of the curve group[5] are generated and denoted c_1, c_2, c_3 and c_4. The coefficient_points are the elliptic curve points C_0, C_1, C_2, C_3 and C_4, corresponding to $c_0, c_1, \cdots$.
- **point_values** - The polynomial $f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4$ is evaluated[6] at $x = 1$, $x = 2$ and so on to generate six secret values $v_1, v_2, \cdots, v_6$ that will be sent to the 6 executors.

  The elliptic curve points, $V_1, \cdots, V_6$ corresponding to these secret values are published as the point_values.

  Each relay can check that the coefficient_points and the point_values are consistent, by constructing the polynomial with point coefficients: $F(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4$ and checking that $F(1) = V_1$, $F(2) = V_2$, and so on.
- **secrets_xor_shared_secrets** - Each secret, v_i, is communicated via a shared_secret.

  The joining relay constructs the shared_secret by multiplying the secret by the executor's relay_pubkey.

  The executor constructs the same point by multiplying his or her private key by the relevant point_value, namely V_i. Having constructed the point, it is serialized and hashed to get the shared_secret.

  The joining relay xor's the shared_secret with the secret and adds the result to the vector of secrets_xor_shared_secrets.

  The executor xor's the shared_secret with the corresponding secret_xor_shared_secret to get the secret, v_i.

If a relay becomes unresponsive, each functioning executor will send a SuccessionMessage:

SuccessionMessage		
Field Name	**Data Type**	**size in bytes**
dead_relay_join_hash	uint160	20
successor_join_hash	uint160	20
executor	Point	34
succession_secret_xor_shared_secret	CBigNum	32
signature	Signature	64

[5]Secp256k1 in this case.
[6]modulo the modulus, of course

where:

- **dead_relay_join_hash -** The hash of the RelayJoinMessage of the departed.
- **successor_join_hash -** This identifies the successor, who is the recipient of the enclosed secret.
- **executor -** This is the relay_pubkey of one of the original 6 executors from the dead relay's RelayJoinMessage.
- **succession_secret_xor_shared_secret -** In this context, the succession_secret, v_i is actually just a part or share of the full succession_secret needed by the successor.

  The shared_secret is constructed by the executor by multiplying the succession_secret by the successor's relay_pubkey.

  The successor constructs the same shared_secret by multiplying his or her private key by the corresponding point_value, V_i, from the original DistributedSuccessionSecret.

  The executor constructs the succession_secret_xor_shared_secret by xoring the succession_secret by the shared_secret.

  The successor reconstructs the succession_secret by xoring the shared_secret with the succession_secret_xor_shared_secret.

  The successor also validates the succession_secret, v_i, by multiplying it by the group generator and comparing the result to V_i.
- **signature**

Having received five out of six of these SuccessionMessages, the successor will know five values, $f(i) = v_i$, of the polynomial, f, which has five unknown coefficients. The five values are used to determine the five coefficients, and the 0^{th} coefficient, c_0, is the succession_secret that can be added to the successor_secret to recover the dead relay's private key.

Of course one message is better than six, so if the relay goes offline gracefully, it will send a RelayLeaveMessage:

RelayLeaveMessage		
Field Name	**Data Type**	**size in bytes**
relay_leaving	Point	34
successor	Point	34
succession_secret_xor_shared_secret	CBigNum	32
signature	Signature	64

where the same shared_secret and xor scheme is used by the relay who is leaving to communicate the succession_secret to the successor in a single number.

One complication is that, if every relay's successor and executors precede that relay in the numbered list of relays, then it means that an attacker who gains complete control of all relay's public keys at one time will automatically be able to reconstruct the private keys of any new relay that joins.

That is, if a malicious entity controls all existing relays, then it will know all the executor secrets and the successor secret because it will control the successors and the executors.

To prevent this, every sixtieth relay has an assigned successor in the future, fifty-nine relays later. In those cases, the joining relay can't send the successor_secret until later, when the successor joins.

To send that secret, the relay sends a FutureSuccessorSecretMessage:

FutureSuccessorSecretMessage		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
successor_join_hash	uint160	20
successor_secret_xor_shared_secret	CBigNum	32
signature	Signature	64

where:

- **credit_hash** - The hash of the credit mined by the relay who joined 59 relays ago and is now sending this message.
- **successor_join_hash** - This identifies the successor, who is the recipient of the enclosed secret.
- **successor_secret_xor_shared_secret** - Communication of the successor_secret via the usual scheme.
- **signature**

8 Complaints and Refutations

The cryptographic schemes used here have the property that, whenever a secret, s, is communicated, the corresponding elliptic curve point, S, is sent along as well[7]. This allows each recipient of a secret to check that the secret is correct, by multiplying s by G, the group generator, and comparing the result to S.

If the two do not match, the recipient can send a complaint. It would be desirable for the complaint to contain a proof that the secret sender did not send the actual secret. Unfortunately, it is not clear how the recipient can construct such a proof without revealing his or her private key.

The sender, however, can indeed produce a proof that the sent secret is correct. This is done merely by publicly revealing the value of s. Any checker can then multiply s by the recipient's public key to obtain the shared_secret.

Because of this, the system used to detect bad secrets uses two components:

- If a recipient receives a bad secret, he or she sends a complaint which merely affirms but does not prove that the secret received, s, does not match the specified point, S.
- Presuming that the recipient's complaint is false - i.e. the correct secret was indeed sent, the sender will send a refutation message containing the secret s in plaintext. Each checker can then verify that the recipient should not have complained.

It is the sender's responsibility to refute any complaints made about secrets sent by that sender. If no refutation is received within a specified time, the complaints are assumed to be valid.

8.1 Complaints About Secrets Used in Trading

The first secrets sent during a trade are those enclosed in OrderCommits, AcceptCommits and SecretDisclosureMessages. Relays who receive the secret parts in the enclosed DistributedTradeSecret objects can complain about the secrets that they receive.

These secrets are always sent in pairs. There's a credit_secret, which is communicated via a credit_secret_xor_shared_secret, and a currency_secret, which is communicated via a currency_secret_xor_hash_of_credit_secret.

[7]Specifically, to communicate a secret, s, to a recipient with the public key K, the ciphertext sent is $(S, s \oplus h(sK))$, where h is a hash function and $\oplus$ is xor. The recipient uses the fact that $sK = kS$, where k is the recipient's private key, to recover the shared_secret $h(sK)$ and then s.

Accordingly, there are two possible failures when recovering the secrets: Either xoring the shared_secret with the credit_secret_xor_shared_secret doesn't give the correct credit_secret, or it does give the correct credit_secret, but the hash of that credit_secret, xored with the currency_secret_xor_hash_of_credit_secret, doesn't give the correct currency_secret.

In either case, the relay will send a RelayComplaint:

RelayComplaint		
Field Name	**Data Type**	**size in bytes**
message_hash	uint160	20
bad_secret_side	uint8_t	1
position	uint32_t	4
bad_secret	CBigNum	32
signature	Signature	64

where:

- **message_hash** - This is the hash of the message containing the bad secret.
- **bad_secret_side** - This identifies whether the bad secret was a credit secret (BID) or a currency secret (ASK).
- **position** - This identifies which of the 34 secrets was bad.
- **bad_secret** - If it's the currency_secret that couldn't be recovered, the relay can (and must) reveal the credit_secret, which allows checkers to verify that the credit secret is good and the currency secret is bad.
- **signature**

If a trader who has sent good secrets sees that a relay has sent a complaint, he or she will respond with a RefutationOfRelayComplaint message:

RefutationOfRelayComplaint		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
good_credit_secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash** - This is the hash of the complaint being refuted.
- **good_credit_secret** - Revealing the true value of the credit secret allows checkers to reconstruct the trade_secret and use that to confirm that the correct values were originally sent for the credit_secret_xor_shared_secret and the currency_secret_xor_hash_of_credit_secret.
- **signature**

The trader will not go through with the trade if a complaint is sent, even if it is subsequently refuted; either case would imply that at least one of the relays is dishonest. The refutation does serve a purpose, though - the relay who raised the invalid complaint will be disqualified from handling further secrets and will not receive a fee at the end of the diurn.

The next secrets sent during a trade are those sent either in CounterPartySecretMessages (sent by traders) or TradeSecretMessages (sent by relays).

If the secret contained in a CounterPartySecretMessage is bad, the trader who receives the message will send a CounterPartySecretComplaint:

CounterPartySecretComplaint		
Field Name	**Data Type**	**size in bytes**
message_hash	uint160	20
side	uint8_t	1
signature	Signature	64

where:

- **message_hash -** This is the hash of the CounterPartySecretMessage containing the bad secret.
- **side -** This is the side of the secret sender (BID for the buyer, ASK for the seller).
- **signature**

For these complaints, it isn't necessary to send a refutation. At least one of the traders is being dishonest. The relays will simply send the secrets forward (i.e. complete the trade). If a system for punishing traders (e.g. loss of a deposit, cryptocurrency address blacklisting) were to be put in place, it would be useful to know which trader is being dishonest, in which case a refutation should be sent.

If a secret in a received TradeSecretMessage is bad, the trader will respond with a TraderComplaint message:

TraderComplaint		
Field Name	**Data Type**	**size in bytes**
message_hash	uint160	20
side	uint8_t	1
signature	Signature	64

where:

- **message_hash -** This is the hash of the TradeSecretMessage containing the bad secret.
- **side -** This is the side of the bad secret (BID for the credit secret, ASK for the currency secret).
- **signature**

If the complaint is not refuted, the relay who sent the TradeSecretMessage will be disqualified, and a backup relay or successor will send the corresponding secret if necessary.
If the original TradeSecretMessage contained a good secret, the relay will reveal that secret:

RefutationOfTraderComplaint		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
good_secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash -** This is the hash of the complaint being refuted.
- **good_secret -** This is the value of the secret which can be used to verify that the originally sent secret was correct.
- **signature**

Although each trader can provoke the relays to reveal secrets by sending complaints, this does not help either trader steal any money, because a trader can only use this mechanism to reveal secrets that he or she already knows.

8.2 Complaints About Relay Inheritance Secrets

Each relay, when joining, must send a secret to his or her successor (except for every 60th relay, whose successor isn't assigned yet), and must also send secrets to six executors.

If the successor receives a bad secret, the successor will complain:

JoinComplaintFromSuccessor		
Field Name	**Data Type**	**size in bytes**
join_hash	uint160	20
signature	Signature	64

where:

- **join_hash** - This is the hash of the RelayJoinMessage containing the allegedly bad successor_secret.
- **signature**

If the successor_secret was actually good, the joining relay will send a RefutationOfJoinComplaintFromSuccessor:

RefutationOfJoinComplaintFromSuccessor		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash** - This is the hash of the JoinComplaintFromSuccessor.
- **secret** - The value of the successor_secret.
- **signature**

Similarly, an executor may complain:

JoinComplaintFromExecutor		
Field Name	**Data Type**	**size in bytes**
join_hash	uint160	20
position	uint32_t	4
signature	Signature	64

where:

- **join_hash** - This is the hash of the RelayJoinMessage containing the bad secret.
- **position** - This specifies which of the six secrets is allegedly bad.
- **signature**

and the joining relay can respond:

RefutationOfJoinComplaintFromExecutor		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash -** This is the hash of the JoinComplaintFromExecutor.
- **secret -** This is the true value of the secret.
- **signature**

These complaints and refutations affect whether or not the relay gets to join successfully. In the case when a relay's successor joins later than the relay, the relay sends that successor the successor_secret in a FutureSuccessorSecretMessage. If the successor_secret is bad, the successor will send a ComplaintFromFutureSuccessor:

ComplaintFromFutureSuccessor		
Field Name	**Data Type**	**size in bytes**
future_successor_msg_hash	uint160	20
signature	Signature	64

where:

- **future_successor_msg_hash -** This is the hash of the FutureSuccessorSecretMessage.
- **signature**

Obviously it's too late by this time to prevent the earlier relay from joining, but the relay can be disqualified if the ComplaintFromFutureSuccessor is not refuted.

RefutationOfComplaintFromFutureSuccessor		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash -** This is the hash of the ComplaintFromFutureSuccessor.
- **secret -** The true value of the successor_secret.
- **signature**

Finally, secrets are sent to a relay's successor when that relay becomes unresponsive. If one of those secrets is bad, the successor will send a SuccessionComplaint:

SuccessionComplaint		
Field Name	**Data Type**	**size in bytes**
succession_msg_hash	uint160	20
signature	Signature	64

where:

- **succession_msg_hash** - This is the hash of the SuccessionMessage containing the bad secret.
- **signature**

If the secret was actually good, the executor will send a RefutationOfSuccessionComplaint:

RefutationOfSuccessionComplaint		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

where:

- **complaint_hash** - This is the hash of the SuccessionComplaint.
- **secret** - This is the true value of the secret sent by the executor.
- **signature**

Again, any checker can use the enclosed secret to verify that the correct secret was sent by the executor to the successor.

Without a valid refutation, the SuccessionComplaint will be taken to be valid and the executor will be disqualified, with his or her duties being inherited by his or her successor.

9 Deposit Addresses

Flex also makes it possible to create a deposit address whose private key is not known by anybody, but which can be retrieved from the network by the authorized deposit-holder. The first stage is for the depositor to send a DepositAddressRequest

DepositAddressRequest		
Field Name	**Data Type**	**size in bytes**
depositor_key	Point	34
curve	uint8_t	1
currency_code	vch_t	3
proof_of_work	TwistWorkProof	variable
signature	Signature	64

where:

- **depositor_key** - This is the secp256k1 public key which will be authorized to transfer or withdraw the private key for the deposit address after it's created.
- **curve** - This specifies the elliptic curve of the deposit address (secp256k1, curve25519 or ed25519).
- **currency_code** - The (usually) 3-letter code for the currency, such as BTC or FLX.
- **proof_of_work** - Because deposits and withdrawals involve a flurry of messages, the depositor must provide a proof of work with each request to make it difficult to overload the system. A fee in FLX would also be possible, but many users may have other cryptocurrencies but no FLX.
- **signature** - The request should be signed with the depositor key.

After the request has been sent throughout the network, the miners add it to the queue of messages to be encoded in the next batch. That is, it gets included in the next mined credit message's hash_list.

When the mined credit which includes the request appears, the hash of that mined credit is combined (using xor) with the hash of the deposit address request. The result is used as a relay chooser, which seeds a random number generator which selects the relays for this deposit address request from the set of relays in the relaystate whose hash is encoded with that mined credit.

Each chosen relay then sends a DepositAddressPartMessage which communicates a part (out of n) of the deposit address's public key to the depositor and breaks the corresponding part of the private key into 4 parts, any 3 of which are needed to recover the part of the private key:

DepositAddressPartMessage		
Field Name	**Data Type**	**size in bytes**
address_request_hash	uint160	20
position	uint32_t	4
address_part_secret	AddressPartSecret	460
signature	Signature	64

where:

- **address_request_hash -** This is the hash of the deposit address request to which this message is a response.
- **position -** This identifies which of the n relays is sending this message.
- **address_part_secret -** This structure, described below, contains one of the n parts needed to reconstruct the private key for the deposit address.
- **signature**

The AddressPartSecret contains:

AddressPartSecret		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
relay_chooser	uint160	20
relay_list_hash	uint160	20
coefficient_points	vector<Point>	$3 * 34$
point_values	vector<Point>	$4 * 34$
secrets_xor_shared_secrets	vector<CBigNum>	$n * 32$

where:

- **credit_hash -** This is the hash of the mined credit which encoded the deposit address request.
- **relay_chooser -** This is the xor of the credit hash with the hash of the deposit address request. It's used as a seed to choose the relays.
- **relay_list_hash -** The hash of the list of chosen relays.
- **coefficient_points -** A randomly-chosen polynomial, $P(x) = a_0 + a_1x + a_2x^2$, is used to generate the secret values, $P(1)$, $P(2)$, $P(3)$, and $P(4)$, which are given to 4 relays. The coefficient points are A_0, A_1, and A_2. A_0 is the elliptic curve point corresponding to $P(0)$, which is this relay's part of the private key for the deposit address.

- **point_values** - These are the elliptic curve points, V_1, V_2, V_3, and V_4 corresponding to the secret values $P(1)$, $P(2)$, $P(3)$, and $P(4)$. Each recipient can verify that $A_0 + A_1 x + A_2 x^2 = V_x$ for x in $\{1, 2, 3, 4\}$.
- **secrets_xor_shared_secrets** - These communicate the secret values, $P(1)$, $P(2)$, $P(3)$, and $P(4)$, to the four relays who receive the parts of this part of the private key. The shared_secret consists of the hash of $P(m)R_m$, where R_m is the m^{th} relay's public key. That relay constructs the shared_secret as the hash of $r_m V_m$, which is then xored with a secret_xor_shared_secret to recover the secret.
- **signature**

Having received all n of the DepositAddressPartMessages, the depositor (and everyone else) adds up the A_0 points (i.e. the elliptic curve points corresponding to the n parts of the secret) to construct the public key of the deposit address. If there are no complaints, described below, from relays about bad secrets, the depositor can send funds to the deposit address.

DepositAddressPartComplaint		
Field Name	**Data Type**	**size in bytes**
part_msg_hash	uint160	20
number_of_secret	uint32_t	4
signature	Signature	64

The number_of_secret field indicates which of the 4 parts of the secret sent in the DepositAddressPartMessage is the one being complained about. If the correct secret was actually sent, the sender will respond with a refutation:

DepositAddressPartRefutation		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

9.1 Disclosures

DepositAddressPartDisclosure		
Field Name	**Data Type**	**size in bytes**
address_part_message_hash	uint160	20
disclosure	AddressPartSecretDisclosure	1
signature	Signature	64

where:

- **address_part_message_hash** - This identifies the DepositAddressPartMessage whose secrets are to de disclosed to a second set of relays in this message.
- **disclosure** - This contains the secrets to be disclosed.
- **signature** - The request should be signed with the same key as the DepositAddressPartMessage.

and the AddressPartSecretDisclosure contains:

AddressPartSecretDisclosure		
Field Name	**Data Type**	**size in bytes**
credit_hash	uint160	20
relay_chooser	uint160	20
relay_list_hash	uint160	20
secrets_xor_shared_secrets	vector<CBigNum>	$n * 32$

The credit hash in this case is the hash of the credit after the one which encodes the request. This is xored with the request hash to generate the relay chooser for the disclosure.

The relays who receive the parts of the sender relay's secret in the AddressPartSecretDisclosure use it to check the secrets sent in the DepositAddressPartMessage. If either the secret in the disclosure is bad or the secret in the original part message was bad, the recipient of the disclosure will send a DepositDisclosureComplaint:

DepositDisclosureComplaint		
Field Name	**Data Type**	**size in bytes**
disclosure_hash	uint160	20
number_of_secret	uint32_t	4
secret	CBigNum	32
signature	Signature	64

The number_of_secret specifies which of the 4 secrets, $P(1)$, $P(2)$, $P(3)$ or $P(4)$, was bad. If the disclosure recipient received a good secret (but the original version of that secret was bad), that secret will be contained in the secret field. Otherwise that field will be zero.

To refute the complaint the sender of the AddressPartSecretDisclosure sends a DepositDisclosureRefutation:

DepositDisclosureRefutation		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

9.2 Transfers

The user who is authorized to transfer the deposit does so by sending a DepositTransferMessage:

DepositTransferMessage		
Field Name	**Data Type**	**size in bytes**
deposit_address	Point	34
previous_transfer_hash	uint160	20
sender_key	Point	34
recipient_key_hash	uint160	20
signature	Signature	64

where:

- **deposit_address -** This is the public key of the deposit address obtained by summing the A_0 points from the part messages.
- **previous_transfer_hash -** If the address has never been transferred before, this will be zero; otherwise, it's the hash of the latest valid transfer.

- **sender_key** - If there has never been a transfer of this address before, this should equal the depositor_key from the deposit request. Otherwise, it should be a key whose hash is equal to the recipient_key_hash from the last transfer.
- **recipient_key_hash** - This is the hash of the key which is, henceforth, authorized to transfer and withdraw this deposit.
- **signature** - The message should be signed with the sender key.

Once the transfer has been sent, the responding relay, which is usually the first relay from the n relays chosen to handle the deposit request, will send an acknowledgement:

TransferAcknowledgement		
Field Name	**Data Type**	**size in bytes**
transfer_hash	uint160	20
disqualifications	vector<pair<uint160,uint160>>	$d * 40$
signature	Signature	64

where:

- **transfer_hash** - This is the hash of the DepositTransferMessage being acknowledged.
- **disqualifications** - The only thing that a responding relay can do wrong is to acknowledge two (or more) contradictory transfers - i.e. to approve a transfer of a deposit to one recipient and also to acknowledge a transfer with the same previous_transfer_hash to a different recipient[8].

  In the case when this does occur, the relay's private key is known to be in the hands of a wrongdoer, and the proof of this is the two conflicting acknowledgements. The next relay in the list of n then acquires the authority to respond to transfer messages, and this relay includes the hashes of conflicting acknowledgements signed by earlier responding relays in the acknowledgements.
- **signature** - The message should be signed with the key of the first non-disqualified relay in the list of n.

When a node receives a TransferAcknowledgement, that node will no longer relay DepositTransferMessages which conflict with the acknowledged transfer. This means that, after the responding relay has sent the acknowledgement to his or her peers, they will no longer give that relay any conflicting transfers. So if a second, conflicting, transfer is acknowledged by the responding relay, that relay must have generated the conflicting transfer himself or herself.

The conflicting acknowledgement, if it occurs soon enough after the original acknowledgement, *is* propagated throughout the network, with the conflicting transfer, so that everybody gets to see that the responding relay is now disqualified.

However, if a certain amount of time passes after the first acknowledgement (currently set to 8 seconds), conflicting acknowledgements will no longer be relayed. This is because, after that time has elapsed, it's extremely unlikely that the second acknowledgement was received before the first by anybody else in the network[9].

[8]The relay could refuse to respond at all, but in that case the relay's successor would take over his or her duties and the acknowledgement would arrive signed by the responding relay's private key.

[9]Nodes may not agree on the order in which messages are received due to delays in propagating these messages throughout the network. We can be confident that no nodes which are properly connected to the network have received two messages in a different order if the time interval between the arrival of two messages is greater than twice the time needed for a single message to propagate throughout the entire network.

9.3 Withdrawals

When the deposit holder wants to retrieve the parts of the deposit address's private key, he or she sends a WithdrawalRequestMessage:

WithdrawalRequestMessage		
Field Name	**Data Type**	**size in bytes**
deposit_address	Point	34
previous_transfer_hash	uint160	20
withdrawal_key	Point	34
signature	Signature	64

where:

- **deposit_address -**
- **previous_transfer_hash -** If the address has never been transferred before, this will be zero; otherwise, it's the hash of the latest valid transfer.
- **withdrawal_key -** If there has never been a transfer of this address before, this should equal the depositor_key from the deposit request. Otherwise, it should be a key whose hash is equal to the recipient_key_hash from the last transfer.
- **signature -** The message should be signed with the withdrawal key.

The relays from the original list of n who are still online will then each send a WithdrawalMessage:

WithdrawalMessage		
Field Name	**Data Type**	**size in bytes**
withdraw_request_hash	uint160	20
position	uint32_t	4
secret_xor_shared_secret	CBigNum	32
signature	Signature	64

If any of the sent secrets are bad, the withdrawer will send a WithdrawalComplaint:

WithdrawalComplaint		
Field Name	**Data Type**	**size in bytes**
withdraw_msg_hash	uint160	20
signature	Signature	64

and if the complaint is without merit, the relay can send a refutation:

WithdrawalRefutation		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64

Some of the original relays may have disconnected since the deposit address was created. If any WithdrawalMessages don't arrive, each of the 4 relays who originally received parts of the corresponding part of the secret will send a BackupWithdrawalMessage:

BackupWithdrawalMessage		
Field Name	**Data Type**	**size in bytes**
withdraw_request_hash	uint160	20
secret_revelation_message_hash	uint160	20
position	uint32_t	4
secret_xor_shared_secret	CBigNum	32
signature	Signature	64

where the secret_revelation_message_hash is the hash of the original DepositAddressPartMessage.

If any of those 4 relays fail to repond, the corresponding relays who received the same secrets in the DepositAddressPartDisclosure will send a BackupWithdrawalMessage with the secret_revelation_message_hash set to the hash of the disclosure.

If *those* relays fail to respond, their duties will be inherited by their successors, who can recover the secrets and send them to the withdrawer.

Like the withdrawal message, the BackupWithdrawalMessage could contain bad secrets, in which case the withdrawer will respond with a BackupWithdrawalComplaint:

BackupWithdrawalComplaint		
Field Name	**Data Type**	**size in bytes**
backup_withdraw_msg_hash	uint160	20
signature	Signature	64

and the relay sending the BackupWithdrawalMessage can refute the complaint by sending a BackupWithdrawalRefutation:

BackupWithdrawalRefutation		
Field Name	**Data Type**	**size in bytes**
complaint_hash	uint160	20
secret	CBigNum	32
signature	Signature	64