# Flex - Send Coins by Sending Secrets

Peer Node
peernode@gmx.com
November 2015

**Abstract**

Cryptocurrencies can be transmitted by sending secrets instead of signing transactions.

By breaking a secret into many parts, it is possible to effectively use many untrusted third parties to replicate the functions of a single trustworthy third party.

With an appropriate network of such third parties, private keys or parts of them can be securely stored in and retrieved from the network without fear that the keys will be lost or that a single entity other than the rightful owner will be able to reconstruct them.

Flex is a decentralized exchange for cryptocurrencies and fiat money, which uses these capabilities for secure cryptocurrency trading.

In addition to trading, cryptocurrencies and digital assets can be deposited at an address whose private key is distributed among the network of third parties. Ownership, namely the right to retrieve the parts of the key from the network, can be transferred irreversibly in seconds, providing a rapid alternative to traditional cryptocurrency transactions.

## Contents

## Introduction

Knowledge of a single number, namely the private key, is what's needed to spend a cryptocurrency. The fastest and simplest way to give somebody some bitcoin, for example, is to send them the private key needed to spend it right away.

The problem with this as a payment system is that the person who sends the private key still knows it. After the recipient gets the key, both the recipient and the sender can spend it.

A way to get around this is to express the private key as a combination, such as a sum, of two or more parts. If the key is the sum of two parts, $k = a + b$, and there are two people who each know a part, then one person can give the coins to the other by sending the part that he or she knows.

How do we arrange for coins to be located at a key made up of two parts, each known to a different person? That's easy, and the way to do it will be explained in section I.

What's more difficult is to use secret-passing techniques to securely trade cryptocurrencies. The traders need a solution to the problem: Which trader sends his or her money first, and what happens if the other trader then decides to abscond?

Section II explains how to do it when a trusted third party is available. Without a trusted entity, we look at how to use multiple untrusted entities along with secret-splitting to simulate a trustworthy entity in sections III to VI.

The complete sequence of communications involved in a secure trade involving cryptocurrencies is described in section VII, while section VIII shows how fiat can be traded within the system.

Section IX describes the messaging used to create, transfer, and withdraw from, deposit addresses whose key is stored throughout the network and which no single individual can spend from.

Cryptocurrency mining is the method used to select the third parties, and the details of the mining procedure used are expounded in sections XI and XII, while sections XIV and XV describe compact structures which Flex uses instead of a blockchain, so that new users can start quickly without needing to download every historical transaction.

Sections XVI to XVIII look at what a future in which this technology is deployed might look like.

Thanks are due to all who have produced advances in cryptocurrency and our understanding of it. Section XIX tries to do justice to them.

## I. Numbers, Points, Bitcoin Addresses

Suppose Alice chooses a number and Bob chooses a number and they keep these secret from each other. How do we arrange it so that somebody who knows both numbers can spend some bitcoins?

For that, we need to know what a bitcoin address is. The familiar string of too many letters and numbers is the output of a hash function, which is designed to output apparently random data. What's more interesting is what's fed into the hash function.

Bitcoin uses elliptic curves as the underlying technology for signing transactions. Elliptic curves are collections of points which satisfy an equation, where each point, $P$, on the curve has an $x$ value and a $y$ value, and $x$ and $y$ are related by the equation of the curve[1].

In that sense, an elliptic curve is like a circle, which also is a collection of points $P = (x, y)$, with $x$ related to $y$ via an equation, which could look like $x^2 + y^2 = a$.

[1] Frequently of the form $y^2 = x^3 + ax + b$.

[2] We can understand it by seeing that we can add points together on a circle - by adding angles. We just need to pick a reference point on the circle, and we can call it twelve o'clock. The angle of a point is the angle between the lines joining it and twelve o'clock to the center of the circle. Then we can add points by adding angles. One o'clock plus two o'clock equals three o'clock, and so on. The rule for adding points on elliptic curves is different: If a straight line intersects the curve at $A$, $B$ and $C$, then $A + B + C = 0$.

The key fact about elliptic curves is that you can *add* points together and get another point on the curve.[2] So you can add a point to itself and get a new point, $P + P$, which we can write as $2P$, and we can also generate $P + P + P = 3P$ and so on.

So you can multiply a point by any positive integer. We pick a point on the curve, call it the generator and denote it by $G$. Then there's a whole series of points, $2G, 3G, \cdots$ that we can calculate and add together: $2G + G = 3G$.

What's very difficult is for somebody to figure out what $n$ is if they are given $G$ and the point $nG$. They would be given two sets of coördinates, $(x_1, y_1)$ and $(x_2, y_2)$, and asked to say how many times $(x_1, y_1)$ needs to be added to itself to yield $(x_2, y_2)$. There's no known efficient way to do this, and when $n$ is very large, this is impractical with modern computers.

With bitcoin[3], $n$ is the private key, and $nG$ is the public key. Somebody who knows $n$ can produce digital signatures that can be verified by anybody who knows $nG$, i.e. who knows the numbers $x_2$ and $y_2$.

Those digital signatures are the signatures of bitcoin transactions. Bitcoins that you own are "at" a public key $nG = (x_2, y_2)$, whose private key, $n$, you know.

If somebody else knows the number $m$, that person can give you the point $mG$ and ask you to send bitcoins to it. You can do so by using $n$ to sign a transaction specifying that some quantity of the bitcoins at $nG$ are henceforth under the control of the person who knows the key corresponding to the (hash of the) point $mG$.

A common notation, which we will use, is to refer to numbers[4] using lower case letters, $a, b, \cdots$ and the corresponding points using upper case letters: $A = aG$.

Now back to Alice and Bob. Equipped with our understanding of elliptic curves, we can present the following solution to the problem of how to require both Alice's and Bob's secret in order to spend the bitcoins:

1. Alice picks a random number, $a$, and publishes $A = aG$.
2. Bob picks a random number, $b$, and publishes $B = bG$.
3. We compute the point $A + B$ and send bitcoins to it.

After that, in order to spend the bitcoins, somebody would need to know the value of $a + b$.

So we've created a condition in which Alice can give the bitcoins to Bob by sending $a$ to him, and Bob can give the bitcoins to Alice by sending $b$ to her.

Upon receiving $b$ from Bob, Alice can spend the bitcoins immediately, so this "transaction" is close to instant, and is irreversible.

Next, we'll see how these tricks can be used for cryptocurrency trading.

## II. A Simple Trade

Suppose that Alice and Bob are traders. Let's say Alice has litecoins which she wants to trade for Bob's bitcoins.

Let's also suppose that Eddie is a third party trusted by both Alice and Bob.

The following protocol is a three-person version of the trick used in the previous section:

1. Alice chooses random numbers, $a$ and $c$, then sends $a$ to Eddie and $c$ to Bob.
2. Alice publishes $A$ and $C$, the elliptic curve points corresponding to $a$ and $c$.
3. Bob chooses a random number, $b$, then sends $b$ to Eddie and also publishes $B$.

After this:

- Everybody knows $A$, $B$, and $C$.
- Alice knows $a$ and $c$.
- Bob knows $b$ and $c$.
- Eddie knows $a$ and $b$.

[3] and elliptic curve cryptography in general

[4] or, more exactly, field elements

If coins are now sent to the public key $A + B + C$, then Eddie can give the coins to Alice by sending $b$ to her, or to Bob by sending $a$ to him.

To complete the trade, two such addresses are created, say $A + B + C$ and $D + E + F$. Alice sends litecoins to one of these and Bob sends bitcoins to the other. When the coins have arrived, Eddie gives Alice's coins to Bob and Bob's coins to Alice.

If something goes wrong, for example if either Alice or Bob fails to send any coins, Eddie sends Bob the secret he needs to get back his bitcoins (if he sent them to the address) and sends Alice the secret needed to get back her litecoins (if she sent them).

This is a scheme for trading, but in the absence of a trusted third party, something more complex is needed. In the next section we'll look at how to replace trusted Eddie with a collection of untrusted intermediaries.

Before doing so, it's good to note that with this simple three-person scheme, Eddie doesn't need to do anything if Alice and Bob are acting in good faith.

If Alice sends the litecoins to $A + B + C$ and Bob sends the bitcoins to $D + E + F$, then afterwards, Alice can give the litecoins to Bob by sending $a$ to him, and Bob can send Alice the bitcoins by sending $e$ to her.

So we can use the convention that Alice and Bob complete the trade without Eddie's help, and Eddie steps in only if one or the other party fails to courteously send along the relevant secret.

This means that for most trades, Eddie's mere presence and the belief that he will act correctly is enough to make the trade go smoothly, and Eddie doesn't need to actually do anything. This is convenient, because Eddie is replaced by many entities in the next section, and the last thing we need in a peer-to-peer network is large numbers of entities sending messages for simple trades.

## III. Relays

We are going to use the word *relays* to refer to the multiple entities which, together, form a subtitute for a trusted third party. This is to indicate their reduced authority compared to third parties that are actually trusted. Their only reason for existence is to receive secrets and relay them to the appropriate recipients.

A problem can arise if a large percentage of the relays are controlled by a single entity: That entity could acquire all the parts of a secret, and then recover the secret.

In the trading scheme outlined in the previous section, Eddie didn't have enough information to steal any coins. E.g. he knew $a$ and $b$ but not $c$. He could, however, collaborate with Alice or Bob to defraud the other trader, or he could even be Alice or Bob in disguise.

So in the case of a single entity who controlled all or most of the relays, that entity could pretend to be a trader and defraud his or her counterparty.

The important question in practice is where to find these relays who are willing to but not trusted to send secrets, and what happens if they disconnect or misbehave.

Flex's solution to the question of where to find them is to use cryptocurrency mining to identify a collection of entities who are probably not controlled by the same entity. Whoever mines a block can become a relay and hold some secrets.

The appearance of a new block is random, and we can consider the block-finding process to be a stochastic selection procedure which selects miners in proportion to their mining power. This gives us a collection of entities who individually are not trustworthy, but who are probably not conspiring together.

When we randomly select a large number of miners and entrust each with a secret, the probability that a single entity knows all (or nearly all) of the secrets is very low.

## IV. Secret Splitting

In section II, Eddie didn't choose or send any secrets. He just received secrets, $a$ from Alice and $b$ from Bob.

This means that the procedure doesn't have to be modified much: Alice will send $n$ secrets, $a_1, a_2, \cdots, a_n$, to $n$ different relays and Bob will also send $n$ secrets, $b_1, b_2, \cdots, b_n$, to $n$ relays.

And that's it. Now we have a complete, multi-entity distributed third party ready to efficiently and silently accommodate trades.

Unless something goes wrong, of course. For example, some relays might disconnect during the trade, when the money is stuck at an elliptic curve point whose private key nobody knows. Or they might be malicious and send random numbers instead of the real secrets.

The next section tackles the question of what to do when relays disconnect while holding secrets. The section after that outlines the principles of wrongdoing, accountability, and retributive justice in the society of Flex relays and miners.

The central technology that will be relied on is that of splitting a secret into many parts, so that some minimum number of parts (not necessarily all) are required in order to reconstruct the secret.

It's readily comprehensible that, if the parts of the secret are $a_1, a_2, \cdots, a_n$ and the secret is $a_1 + a_2 + \cdots + a_n$, then all the parts are needed. If you know all the others but not $a_2$, then you can't recover the secret.

There are more complex schemes which permit reconstruction of the entire secret from any $k$ parts out of $n$, for arbitrary positive integers, $n$ and $k < n$. Flex uses the method of interpolating polynomials described in [7][5]. The full details are given in the Flex Protocol document[6].

By requiring, for example, only 5 out of 6 parts in order to reconstruct a secret, a peer-to-peer network can suffer some damage or malicious behaviour and continue to operate successfully. As the next section explains, it's possible to iterate this secret-decentralizing procedure to obtain a network robust enough to withstand up to half of the network being knocked offline at once without any data loss.

## V. Dead Relays and Inheritance

### I. The Inevitability of Dead and Malicious Relays

If, for every trade, each trader breaks a secret into many parts and gives each part to a relay to keep safe, then eventually some of those relays will disconnect while holding secrets needed to complete the trade.

Luckily, the scheme described in section II would be able to work without the loss of any coins even if Eddie were to be knocked offline, provided that Alice and Bob are acting in good faith. Of course, we can't rely on that, so we need a way of making the network robust to disconnection by relays.

The solution which would be most robust to damage would be for all part of all secrets to be shared by all relays, so that there's no data loss when any relay goes offline. This has the drawback that every relay can reconstruct every secret, so instead of having one intermediary who could conspire with Alice or Bob, there would be many.

We also have to accept the fact that an attacker may gain control over a significant fraction of the set of relays, so we need to ensure (or at least, make it extremely probable) that no attacker who controls, for example, a third of the network will be able to obtain enough parts of a secret to reconstruct the full secret.

This means that we have to be very careful when letting more than one relay know a part of a secret. If an attacker controls 10% of the relays and we let two randomly chosen

[5] In a nutshell, given two distinct points on a line, you can draw the line, so if you give $n$ different points on the same line to $n$ different people, then any 2 of them can reconstruct the line. Given any three distinct points on a parabola, the entire parabola is specified, and so on. The intersection of this line, parabola, or other curve with the $y$-axis provides a number that can be constructed only by acquiring the necessary number of parts.

[6] https://peer-node.github.io/flex/files/protocol.pdf

relays know the same part of a secret, then the attacker has a 19% chance of discovering that part[7]. So it's best if we have each secret known by only one relay, unless there's a good reason not to (and we'll see in the next section that sometimes there is a good reason).

## II. Splitting Has Its Limits

But if each part of each secret is known to only one relay, then don't we lose that part if the relay disconnects? Not necessarily: that part could itself be broken into parts, so that many relays could come together and recover it. Then we could break parts of parts into parts and so on.

There are two problems with this. One is that there are too many messages to send. If there are many parts, then there will be many many parts of parts, and the amount of information to be shared increases exponentially as more rounds of split-and-share are used.

Another problem is that if we split a secret known to one relay into two parts and give those parts to two other relays, then we are increasing the probability that an attacker will get that secret, either by controlling the relay who knows it or by controlling both of the others.

## III. Inheritance of Private Keys is Both Robust and Secure

Flex uses another technique to prevent data loss, one which doesn't split each secret known to each relay, but instead splits the relay's private key into parts, and gives these to other relays, so that if the relay goes offline while holding secrets, a group of relays (called "executors") will send the secrets they know to a "successor" relay, who can then recover the dead relay's private key, and can read every secret ever sent to that relay.

The successor then takes over the duties of the deceased. If the successor has also disconnected, then the successor's successor will take over his or her duties.

What if some of the executors have gone offline? The same principle is used here. Each executor has a successor who can take over the executor's duties if needed.

Will this work in practice, or will we find, with our finite collection of relays, that we rarely have enough executors left online and our attempts to resurrect the lost secrets are doomed to failure?

That depends on how many executors we require in order to give the successor the parts he or she needs to recover the private key. Flex requires secret parts from 5 out of 6 executors in order to complete a succession[8].

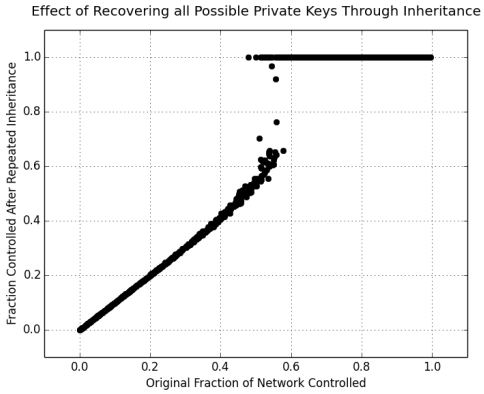


**Figure 1:** *With the key recovery scheme in which 5 out of 6 executors are required to enable a successor to recover a dead relay's private key, an initial collection of more than about 55% can generally recover all keys while an initial collection of less than half can generally not recover more than a few. 1,000 simulations of 600 relays were used to generate the graph.*

Figure 1 shows the results of many simulations, where, in each simulation, a certain number of private keys are initially known,

[7] Made up of a 10% chance that the first relay chosen is controlled by the attacker, plus a 9% chance that the first relay chosen isn't controlled by the attacker and the second relay is.

[8] This is done by expressing the private key as $k = q + s$ where $q$ is a secret sent to the successor and $s$ is broken into 6 parts sent to 6 executors, of which 5 are needed to reconstruct $s$. If 5 executors send their parts of the secret to the successor, the successor can combine those to recover $s$, and add this to $q$ to recover $k$.

and then each key that can be recovered using the 5-of-6 executors plus successor scheme is added to the set of known keys. Then more keys are known so more keys can be recovered and this process repeats until it fails to recover any new keys. The code that produces the graph is included with the Flex software package.

As the figure shows, it's approximately true that if you start with more than half of the keys, you can recover them all, and if you start with less than half then you can't recover very many. There's a region within a few percent of the 50-50 point where it could go either way. Sometimes, 49% will be able to recover everything and sometimes, 51% will fail to recover everything.

But at 40% and 60% it starts to look quite safe. If we're willing to suppose that an attacker trying to recover private keys to steal secrets won't control more than 40% of the network while the honest network itself won't have more than 40% of its relays knocked offline at one time, then the 5-of-6 executors plus successor scheme ensures that data isn't lost due to network damage and also provides little or no benefit to attackers.

## IV. The Line of Succession

How do we choose the successors and the executors? Here we face a problem: If the relays have to split and share their private keys when they join - that is, when they become relays, their their successors and executors will all have joined in the past.

That's a problem for two reasons. One is that, if an attacker manages to obtain control over the entire network at one time, then any new relay which joins will need to give the parts of his or her private key to relays controlled by the attacker, and the attacker will then be able to recover the new relay's key by combining these parts. So an attacker who gains control over the network at one time would get every new relay's private key and so would remain in control forever.

Another undesirable consequence of having every relay's successor be somebody who joined prior to that relay is that it could lead to data loss. Since disconnection occurs silently, and relays who joined longer ago are less likely to still be online, it's quite possible that a relay who joined a long time ago could have no online downstream successors, and there would be no possibility of recovering the relay's private key.

So at least some of the relays will need to have successors that join later than they do. That way, succession can be used to allow newer relays to recover secrets from older relays that have disconnected, and an attacker who gains control of all or most relays will find that some new relays have successors that the attacker doesn't already control, and so the attacker won't be able to inherit all future keys.

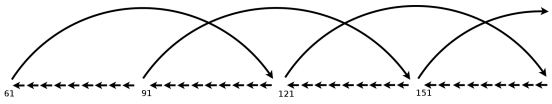


**Figure 2:** *The line of succession used in Flex. The* $120^{th}$ *relay is the successor of the* $61^{st}$*. Subsequent successors are the* $119^{th}$*,* $118^{th}$ *and so on down to the* $91^{st}$*, whose successor is the* $150^{th}$*. Although succession usually passes secrets to an earlier relay, this scheme allows secrets to be propagated forwards in time, helping to prevent data loss.*

In Flex, relays join approximately once a minute, and each relay is assigned a number indicating the order in which they joined. The successors are chosen according to the scheme shown in figure 2. Specifically, each relay's successor is the relay that joined immediately before it, unless the relay's number is of the form $1 + 30n$ for $n > 1$, so the $61^{st}$ relay, the $91^{st}$, the $121^{st}$ and so on all have successors that join in the future. The future successor is always the one that joins 59 places later. The predecessor sends a message containing the secret to the successor when the successor joins.

The executors for each relay are those relays that joined before the relay at set offsets. These offsets are 9, 17, 19, 24, 31 and 36. Relays joining after position 36 split and share their

private keys with their executors and successor; those joining before position 37 don't.

## VI. Bad Relays, Complaints, and Refutations

We not only have to deal with relays that might go offline or try to steal secrets, but we also have to be prepared to deal with relays who might try to interfere with trades or block money by violating the protocol and, for example, sending random data instead of the correct secrets.

To deal with this, we need a system that can detect if the secret sent by a relay is correct. If a relay sends an incorrect secret, and the network knows that the secret sent was bad, the relay's successor can be called upon to take over the relay's duties and send the correct secret.

The problem also exists with secrets that are sent to relays by a trader. What if the trader sends bad secrets? Since nobody other than the recipient knows what was sent, we obviously have to rely on the recipient relay to publish a complaint when a bad secret is received.

But then what if the trader sends good secrets, but one of the relays complains that he or she received a bad secret? How does the rest of the network know who to believe?

## I. Hashes and Ciphers

We need a way of sending secrets so that somebody other than the sender and recipient can audit what was sent. That's what's necessary to resolve a dispute between the sender and the recipient about what was sent in a specific message.

A common way of encrypting with elliptic curve keys is for the sender to xor the secret with something that both the sender and the recipient know, but that others don't know. The recipient performs the same xor operation on the incoming message and recovers the secret.

For example, if Bob's private key is $b$ and his public key is $B = bG$, while Alice's private key is $a$ and her public key is $A = aG$, then they can both construct the point $abG$. Bob constructs it by multiplying his private key $b$ by Alice's public key $A$. Alice multiplies $a$ by $B$.

In practice Bob and Alice would apply a hash function to the point and get a uniformly random string before xoring the result with the message. When using xor to encrypt a secret, it's important that whatever you're xoring the secret with doesn't have any regularities. The $y$ and $x$ coördinates of an elliptic curve point are related by the equation that defines the curve, and the distribution of $x$ and $y$ values themselves may not be uniform.

So with a hash function, $H$, and a secret, $s$, we can describe a primitive cipher which Alice and Bob can use to communicate:

$$m = s \oplus H(abG) \tag{1}$$

where $\oplus$ means xor and $m$ is the encrypted message.

The recipient uses the reverse operation to recover the secret:

$$s = m \oplus H(abG) \tag{2}$$

This encryption scheme doesn't work for us, though, because we need to be able to let somebody audit the message and confirm that $s$ was indeed sent, in the event of a dispute between Alice and Bob.

An auditor could only do so if the auditor knew the value of $H(abG)$. Either Alice or Bob could tell the auditor what string of bytes $H(abG)$ is equal to, but neither Alice nor Bob can *prove* that $H(abG)$ is equal to what they say without revealing their respective private keys. And the auditor will probably not get the same answer from both of them, since one of them is trying to fool everybody about what message was sent.

The solution that Flex uses is to replace the sender's private key with the secret. So suppose Alice is sending the value of $s$ to Bob, then Alice sends:

$$s \oplus H(sbG) = s \oplus H(sB) \tag{3}$$

Alice also sends the "public key" of the secret, which is the elliptic curve point $sG = S$,

so the message sent is:

$$m = (S, s \oplus H(sB)) = (S, s \oplus H(Sb)) \quad (4)$$

Bob can take the value of $S$ from the message and multiply it by his private key, $b$, before applying the hash function to get $H(Sb) = H(sB)$. Then it's a matter of xoring this with the other part of the message, namely $s \oplus H(sB)$ to recover $s$.

## II. Refutations

Equation 4 allows anybody who knows Bob's public key and the secret $s$ to verify that the message $m$ does in fact communicate the value of $s$ to Bob. In the event of a dispute between Alice and Bob about what was sent, Alice can publish the value of $s$ and then everybody else can construct $S$ and $s \oplus H(sB)$ for themselves and see if that's the same as $m$.

Bob can't do that, because Bob's contention is that $s$ wasn't correctly communicated to him. Bob can't prove that he didn't receive $s$; he can only *affirm* it. Alice can prove Bob's affirmation wrong by revealing $s$, and it's reasonable to accept Bob's affirmation if Alice can't do this.

The sequence of events involved in resolving a dispute about a sent secret, $s$, is therefore:

1. The sender sends the message, $m$, which should equal $(S, s \oplus H(sR))$, where R is the recipient's public key.
2. The recipient complains that the secret received was incorrect.
3. Either the sender publishes the correct value of $s$ and $m$ turns out to be equal to $(S, s \oplus H(sR))$, in which case we all know that the recipient was wrong to complain, or the sender doesn't do this and the recipient's complaint is upheld.

In either case, the network knows which secrets need to be resent by successors and who is misbehaving. The message the recipient sends to complain is called a complaint, and the message the sender sends to prove the complaint wrong is called a refutation.

## III. Double Checking Secrets

The system described so far is still not secure against collaboration between the sender and recipient.

For example, what if a trader controls a few of the relays to whom secrets need to be sent? The trader could send bad secrets to those relays and suppress the complaints, leading the rest of the network to perceive that nothing is wrong and to proceed with the transaction. The fact that the correct secrets were never sent wouldn't be discovered until the other trader tries to reconstruct the private key needed to spend the coins and it just doesn't work.

Even if the successors of the bad relays were to step in and read the secrets that those bad relays had received, they would get the bad secrets that the trader sent, and could never recover the real secrets.

To deal with this, Flex resorts to permitting two relays to know each part of each secret[9]. Each relay can check, using the auditing process described in §I, that the other relay received the same secret.

This way, in order to send a bad secret and not get caught, the sender would need to control both relays which receive that secret. An attacker who controls a significant fraction of the network will occasionally control both recipients of a secret, though, so a further defense is needed.

Flex provides this further defense by requiring the sender to reveal the parts of a secret to one set of relays, and then subsequently selecting a second set of relays which the sender could not have been able to predict, and require the sender to reveal the parts of the secret to them.

If the sender controls some of the first set of relays, and intends to send bad data instead of the real parts of the secret, using control over the recipient relay to suppress the complaint,

[9] This is only for secrets broken into parts and distributed throughout the network for the purpose of handling cryptocurrencies. Each part of each relay's private key is given to only one other relay.

the sender will need to do so at a time when it's unknown which relay will be double-checking that secret after the second set of relays is chosen.

Since the attacker is presumed to control less than half of the network, it will in general be likely that the double-checker won't be under the control of the sender. So any attempt by a sender to send bad secrets is likely to result in the sender getting caught by the double-checker.

Flex also permits reconstruction of a secret even when one or more parts are missing. This means that an attacker who tries to send bad secrets would need to need to sneak them past this double-checking mechanism multiple times.

## IV. Penalties

When a relay is caught sending bad secrets or complaining about valid ones, it makes sense to regard that relay as disqualified and not entrust it with any more parts of secrets. Its successor will complete any outstanding duties and then the relay will be skipped in future selections of relays.

It's appropriate, however, to also have an incentive to keep the relays honest. As later sections explain, there are occasions when participating relays are paid a fee for their services. This is forfeited if any relay fails to respond, sends bad secrets or complains about valid ones.

These incentives, plus the system of sending secrets, complaints and refutations described earlier, as well as the key-recovery process, allow us to use a collection of untrusted, unreliable relays to replicate the functions of a reliable and trustworthy party without ever having any entity who knows enough to steal any currency.

# VII. Trading

Now that Eddie from §II has been replaced by a network of untrusted and unreliable relays who are incentivized to detect and remedy each other's misbehavior, we can look at the messages involved in completing a trade.

## I. Credits and Currency

Flex needs its own cryptocurrency for two reasons. One is to provide incentives to attract a network of relays. The other is to provide a native currency for the exchange which is fully visible to the relays.

Relays can't be expected to host every blockchain, and nor can they be expected to connect to specific centralized services to determine whether a cryptocurrency transaction has taken place. Connnecting to those services would reveal the IP addresses of the relays.

With a native cryptocurrency, relays can see transactions and their amounts, and so can confirm that one trader has funded his or her half of the trade.

A third possible possible use would be for fees for trades, transactions or deposits to be charged in the native currency. At present there are no such fees, so users can trade, transact, deposit and withdraw for free. The absence of fees is likely to help encourage adoption of Flex, but if the network is frequently overloaded with traffic in the future, the introduction of fees could be a useful way to reduce traffic.

To reduce confusion, cryptocurrencies other than Flex's native one will be called currencies, cryptocurrencies or coins throughout the rest of this document, while Flex's native cryptocurrency will be called *credits*, or *Flex credits* or FLX.

FLX is the numéraire in the Flex decentralized exchange. Prices for other currencies are alway quoted in FLX, and trades are always between FLX and another crypto or fiat currency. Trades between, for example, bitcoins and litecoins need to be done in two stages, first from bitcoins into credits, and then from credits into litecoins. Future iterations of the exchange could hide this from the users if desired, so that users can simply say that they want to exchange bitcoins for litecoins and the software could handle the parts which involve

FLX without requiring the user to think about it.

Because credits are the reference currency, each trade has a *buyer*, who gives credits and buys the other currency, and a *seller* who sells the other currency and receives credits.

## II. Orders

When Alice sends an order, she needs to include, among other information[10]:

- A public key.
- Whether this is a buy order a sell order.
- The currency code of the currency she wants to buy or sell.
- The amount of currency she wants to buy or sell.
- The price at which she wants to buy or sell.
- The hash of a relay chooser.

The public key is used to sign messages and receive secrets using the cipher described in §VI.I.

The relay chooser is a secret number known only to Alice. It will be combined with a relay chooser chosen by whoever accepts the order (in this case, Bob), in order to provide a number which neither Bob nor Alice can predict in advance. This will then be used to choose the relays that hold the parts of the secrets in place of Eddie.

For the sake of concreteness, let's suppose that Alice want to buy bitcoins.

## III. Accepting Orders

Bob sees Alice's order and decides to accept it. Bob will need to send a message to the network containing, among other information:

- A public key
- The hash of the order being accepted
- A relay chooser

When an order is received by a node in the network, the node will feed the order into the hash function, $H$, and get a 20-byte sequence of characters which uniquely identifies the order. The order is then stored in the database so that it can be retrieved using its 20-byte hash.

This allows each node to retrieve Alice's order when Bob's Accept Order message is received, so that each node knows which order is being accepted by Bob.

Bob publishes his relay chooser with his Accept Order message. At the time when Bob does this, he does not know what Alice's relay chooser is, although he knows its hash. He knows, however, that Alice can no longer change her mind about what her relay chooser is. She can only use the relay chooser whose hash she has already published.

Alice has no control over the relays that will be chosen (since it depends on Bob's relay chooser, which she doesn't know), and Bob has no control over which relays are chosen (since it depends on Alice's relay chooser, which he doesn't know). This allows each trader to be confident that the other trader didn't get to interfere with the choice of relays.

## IV. Committing

When Alice gets Bob's Accept Order message, she sends an Order Commit message which confirms that she's still online and intends to go through with the trade. The message contains, among other information:

- The hash of the Accept Order message.
- Alice's relay chooser.
- Two secrets for Bob, $c$ and $f$.
- Two sets of 34 parts secrets for 34 relays, $a_1, \cdots, a_{34}$ and $d_1, \cdots, d_{34}$.

[10] Digital signature, proof of funds and other information not directly related to secret-passing, relay-choosing and sending cryptocurrency.

- Elliptic curve points for all of the secrets and parts of secrets, including $C$, $F$, $A$ and $D$, where $a$ is the secret that can be recovered by combining any 31 out of 34 of the secrets $a_1, \cdots, a_{34}$, and $d$ is the corresponding secret for $d_1, \cdots, d_{34}$.
- The hash of the list of relays.
- The hash of a second relay chooser.

Obviously the included hash of the Accept order message lets the recipient know what proposed trade is being commited to.

When Alice reveals her relay chooser, everybody can deduce the list of relays, by combining it with Bob's relay chooser and using the result to seed a random-number generator which picks relays from those available. The hash of the relay list is included so that everybody can confirm that they agree on the relays chosen.

Since this is a trade, there are going to be two escrow addresses, one for credits, $A + B + C$, and one for the currency, $D + E + F$.

If sombody learns $a$ and $b$ and $c$, then that person can spend the credits. If somebody learns $d$, $e$ and $f$, that person can spend the currency.

Alice chooses the secret numbers $a$, $c$, $d$ and $f$, corresponding to the elliptic curves points $A$, $C$, $D$ and $F$.

Alice shares $c$ and $f$ with Bob and breaks $a$ and $d$ into lots of parts and spreads those parts throughout the network.

When Alice sends this message, the relays receive their secret parts and Bob receives his. If any of the relays find that the secret sent by Alice is bad, that relay will send a complaint. If Bob finds that a secret is bad, Bob just won't commit to the transaction.

If Bob receives the correct values of $c$ and $f$ from Alice, Bob will send an Accept Commit message, containing, among other information:

- The hash of the Order Commit message.
- Two sets of 34 parts secrets for 34 relays, $b_1, \cdots, b_{34}$ and $e_1, \cdots, e_{34}$.
- Elliptic curve points for all of the secrets and parts of secrets, including $B$ and $E$.
- A second relay chooser.

When the network receives Bob's message, everybody will know the values of $A$, $B$, $C$, $D$, $E$ and $F$, so everybody will know the escrow addresses $A + B + C$ and $D + E + F$. At that point, it is possible (but not necessarily wise) for Alice and Bob to send their funds to those addresses.

## V. Disclosing

In his Accept Commit message, Bob revealed his second relay chooser. This, combined with Alice's second relay chooser, whose hash she published in the Order Commit message, chooses a new set of relays which neither Alice nor Bob can predict in advance[11]. This new set of relays is the second set of relays described in §VI.III, to whom Alice and Bob must reveal their sets of 34 parts of secrets.

These relays can check the secret parts sent to ensure they're good and can also audit the secret parts previously sent to the first set of relays, to ensure that neither Alice nor Bob even tried to send any bad secrets.

To accomplish this, Alice first sends a Secret Disclosure message containing, among other information:

- The hash of the Accept Commit message.
- Alice's second relay chooser.
- Bob's second relay chooser.
- The hash of the list of relays chosen.
- Two sets of 34 parts secrets for 34 relays, $a_1, \cdots, a_{34}$ and $d_1, \cdots, d_{34}$.

Bob then sends one as well:

- The hash of the Accept Commit message.
- Alice's second relay chooser.
- The hash of the list of relays chosen.

[11] We ignore the possibility that Alice and Bob could be conspiring together. Only their money is at stake.

- Two sets of 34 parts secrets for 34 relays, $b_1, \cdots, b_{34}$ and $e_1, \cdots, e_{34}$.

After getting through this sequence of messages, Alice and Bob wait and listen for complaints from relays before proceeding to send credits and coins to the escrow addresses $A + B + C$ and $D + E + F$.

If any relays complain, Alice and Bob abort the trade.

## VI. Payment Confirmations

When the Alice sees that the bitcoins arrived at the escrow address, $D + E + F$, she sends a payment confirmation, which signals to the relays that the trade should go forward.

When Bob sees that the payment confirmation has been sent, he knows that the Alice can't reverse the trade any more. After this, presuming that Alice's credits have arrived at $A + B + C$, the relays will give the credits to Bob (by sending him the $a_1 \cdots a_{34}$) if Alice doesn't send $a$ to him within a specific time limit (or if she does but he complains).

Conversely, the relays will send the coins to Alice (by sending her $e_1 \cdots e_{34}$), if Bob doesn't send $e$ to her.

If the payment confirmation doesn't arrive within a specified period of time, the trade is cancelled and the relays will send the credits back to Alice by sending $b_1 \cdots b_{34}$ to her and the bitcoins back to Bob by sending $d_1 \cdots d_{34}$.

# VIII. Fiat

Fiat can be traded if the network can receive a reliable signal that a specific fiat payment has occurred. This comes in the form of a signature by a key that both traders trust to certify the occurrence of fiat payments.

When trading fiat money, the public key of a trusted notary is specified in the order. Traders should only accept fiat orders if they recognize and trust the key of the notary.

The notary is expected to be a network service which is connected to the Flex network and issues two types of messages:

- Transaction Acknowledgement

This indicates to the traders that the notary is online, recognizes the proposed trade and is ready to respond if given proof of the occurrence of the fiat transaction. The traders shouldn't proceed with the trade if they don't get an acknowledgement from the notary.

- Payment Confirmation

This substitutes for the payment confirmation sent by the fiat buyer. In the case of a fiat trade, the buyer can't be trusted to affirm that he or she received the fiat payment. If the buyer were able to get the credits back as well as receive the fiat payment, then many unscrupulous fiat buyers would do just that.

So before sending any fiat, the fiat seller should ensure that the credits have arrived at the escrow address and that an acknowledgement has been received from the notary. After that, the fiat seller should send the fiat money to the fiat buyer, making sure that the notary receives the proof required to confirm the transaction.

Upon receipt of the notary's payment confirmation, the relays will know that they should release the credits to the fiat seller, should the fiat buyer not do so.

How exactly the notary receives and validates a proof that the fiat payment has occurred is beyond the scope of Flex and will be different for each payment method. Setting up a notary which can receive and understand these proofs is the main task required in order to plug a fiat payment method into Flex.

However, we will briefly look at two options for proving the occurrence of fiat payments.

## I. Instant Payment Notification

Many online fiat wallets and payment processors[12] offer an Instant Payment Notification service, sometimes called a callback. This consists of an http POST message sent by the payment processor to a specified url containing details of the payment.

[12] including OKPay, WebMoney, Perfect Money, BitPay, BitcoinPay, Cryptopay as well as PayPal, Skrill and most others.

If the payment processor provides such a service, and allows the sender to specify the url, then the sender can tell the payment processor to notify the notary when the payment occurs.

To plug one of these payment processors into Flex, a notary service needs to be running which can receive these IPN messages and validate them (i.e. ensure it's really a message from the payment processor and not from somebody faking the notification).

Some systems, like OKPay, make it easy for anybody to validate the notification: The recipient posts the data back to OKPay's servers and they confirm it came from them.

Others send an accompanying signature consisting of a hash of the payment details concatenated with a secret shared between the payment processor and the payer or recipient. Often, these shared secrets are API keys or passwords which provide access to the payer or recipient's account with the payment processor. So these signatures usually cannot be verified unless the buyer or seller grants the notary access to his or her payment processor account, which would enable the notary to spend the money in the account and is therefore unacceptable.

However, it is usually still possible to verify that the notification came from the payment processor's servers, by comparing the incoming IP address against a list of known servers, and many payment processors publish this list so that this check can be performed.

Of course, any fiat payment method which is plugged into Flex must be irreversible. Otherwise, the fiat seller can complain to the payment processor and recover the fiat money after receiving the credits.

## II. Receipts and Pagesigner

Few payment processors provide signed receipts to payers after a payment, but such a feature would make it very easy to incorporate that payment method into Flex. The notary would just need to check the signature and payment details and then could send a payment confirmation.

Most payment processors, however, provide an option to connect using https and also allow the user to view the details of past transactions. A user who uses the PageSigner browser plugin or command-line script [1] can save a file containing a digital proof that a specific website served a specific page.

This digital proof can be used as a receipt if it proves to the notary that the payment processor does report that the transaction occurred.

So in a simple implementation of this protocol, the fiat payer would be asked to go to the transaction history section and click on the relevant transaction and then click the pagesigner button and then save the file and then go to the notary's website and upload the file. Then the notary would send the payment confirmation and the fiat payer would receive the credits.

A more convenient version could involve a script making the fiat payment using an API key; retrieving a JSON representation of the transaction from the payment processor via https and then sending the pagesigner proof of the JSON message to the notary automatically.

With such a script in place, or using the IPN system described in the previous section, fiat payment methods which can be interacted with programmatically can be traded as easily as a cryptocurrency.

## IX. Deposits

The system of trading described in previous sections has the advantage that not even if all of the relays joined together in a conspiracy to steal coins, could they steal the coins of Alice and Bob if they were honestly trading with each other. Alice and Bob shared a secret, $c$, which the relays, and Eddie from §II, never knew, and which was necessary for anybody who wants to spend coins located at $A + B + C$.

We can consider what advantages we would gain if we were willing to forego this one shared secret. There would be coins at an address like $A + B$, and the relays would have enough information together that they could steal the credits and coins if they were

to conspire together.

If we're willing to accept this possibility (we can make the requirements for reconstructing the secret more stringent if necessary) then the coins or credits at the escrow address would have the interesting property that the relays could, collectively, transfer ownership (i.e. the right to receive the parts of the private key) in seconds, without the assistance of the original depositors.

It would also be possible for the relays to construct an escrow address without requiring both traders to be present. Instead of an address like $A + B$, where Alice provides $A$ and Bob provides $B$, an address like $R_1 + R_2 + \cdots + R_n$ could be used, where relays provide the individual parts of the address. Unlike $A$ and $B$, there's no entity at all that knows the value of $r_1 + r_2 + \cdots + r_n$.

A very important requirement is that nobody should be able to control which relays get chosen to create a specific deposit address. If somebody could choose the relays, they could choose relays they control, and steal the secrets.

To satisfy this requirement, it's necessary to have some information which nobody is able to predict. Flex achieves this by mining. When a Deposit Address Request message is sent, the miners add its hash into the snapshot of the network state that they're encoding. The hash of the message which announces the next batch of credits[13], which also contains the hash of the request, is used as the unpredictable information. This is combined with the hash of the Deposit Address Request to select a collection of relays uniquely determined by the request and the batch which encodes it.

An attacker who tries to control the choice of relays by controlling the next batch that appears would need to be able to mine much faster than the network: In order to have several possible sets of relays to be able to choose from, the attacker would need to mine that many valid batches before the real network mines a single one.

Flex uses the following scheme to create deposit addresses:

- A user sends a Deposit Address Request. This specifies a public key which will initially be authorized to request the secrets parts of the private key needed to spend coins from the deposit address.
- When the deposit address request gets encoded into the network state, $n$ relays are chosen to handle the request.
- The $n$ relays[14] choose $n$ secret numbers, $r_1, \cdots, r_n$ and publish the corresponding elliptic curve points, $R_1, \cdots, R_n$.
- Each relay also breaks his or her secret into 4 parts, any 3 of which can reconstruct the secret, and reveals these 4 parts to 4 relays.
- The relays send this information throughout the network in a series of Depost Address Part messages. The deposit address is $R_1 + R_2 + \cdots + R_n$, and the user can construct this address upon receipt of these messages.
- After the next batch appears, another $n$ sets of 4 relays are chosen, and the $n$ relays each send a Deposit Address Part Disclosure message, revealing each of the 4 parts of each secret to another relay.
- The user waits to see if any relays complain about bad secrets, in which case the deposit is cancelled. If there are no complaints, the user can deposit cryptocurrency to the deposit address.

All $n$ secrets are needed to construct the private key, $r_1 + r_2 + \cdots + r_n$. With $n = 12$, it is unlikely that an attacker will control enough of the network to get all 12 secrets.

The double-checking procedure used here and described in §III makes it unlikely that false secrets were sent, and the 3-of-4 secret reconstruction scheme, combined with inheritance of private keys if necessary, makes it

[13] Flex groups credits together into batches, rather than grouping transactions into blocks, as other cryptocurrencies do.
[14] $n$ is currently set to 12.

highly likely that the correct secrets can be retrieved from the network when necessary, even if the original relays who chose the secret parts have disconnected.

## I. Withdrawals

After a user receives a deposit address and sends coins to it, that user is entitled to withdraw the coins. To do this, the user sends a Withdrawal Request message signed with the key which is currently authorized to withdraw the funds.

The $n$ relays will respond by sending $n$ Withdrawal messages, each containing one part, $r_i$, of the private key. Upon receiving all $n$ parts, the user can add them together to recover the private key and import or sweep the private key into his or her wallet.

If all of the original relays are still online, the withdrawal should take only a few seconds. If some have gone offline, or send bad secrets instead of the real ones, the second set of relays who received those secrets can send them on.

Only 3 of these 4 parts need successfully arrive in order for the user to be able to recover the secret part of the private key. The second set of 4 relays to whom the parts of the secret were disclosed can quickly send any of those 4 parts if two or more of the original 4 recipients have disconnected since then.

Any relays which are discovered to have disconnected by this time will pass their duties on to their successors, who can discover and send the secrets needed. Unless the network has suffered massive damage, disconnecting half or more of the relays, we can be confident that all parts of the secret will eventually be sent to the user making the withdrawal.

## II. Transfers

Transactions are slow in a decentralized network because of the need to achieve consensus across the network. The alternative, much faster method is to use a single authority.

The danger which requires us to go slowly in peer to peer networks is double-spending. Somebody could send the same money to two different destinations at once, leaving half of the network thinking one transaction came first and is valid, and the other half thinking that the other transaction came fist.

Flex solves this by appointing a single authority, namely the first relay in the list of $n$ holders of parts of the deposit address's private key, to decide on the sequence of events. A depositor sends a Deposit Transfer message, identifying a new public key, called the recipient key, which is henceforth authorized to transfer or withdraw the deposit.

Then the authorized relay responds with a Transfer Acknowledgement message, which lets the network knows that this, and not a different, transfer of this deposit from one key to another key, took place.

This is not a very powerful authority because it's only in the case when the depositor tries to double spend (or in this case, double transfer) that the relay would even have a choice about what sequence of events took place. If the depositor doesn't double spend, the most damage the authorized relay can do is not respond to Deposit Transfer messages with Transfer Acknowledgments. If that happens, the relay's successor takes over.

But if the first relay on the list is itself the double-spender, then it could simultaneously send two conflicting Transfer Acknowledgements and let these propagate throughout the network.

The defense against this is simply to wait a few seconds after receiving one Transfer Acknowledgement to see if a conflicting one arrives. If it does, first relay on the list (who has been caught signing contradictory Transfer Acknowledgements) is disqualified and the second relay on the list acquires authority for this deposit address, and sends a fresh acknowledgement.

The Transfer Acknowledgement message contains:

- The hash of the transfer being acknowledged.
- A list of disqualifications, where each disqualification contains two hashes which

identify conflicting Transfer Acknowledgements signed by a relay who previously had authority over this deposit address.

- A signature.

If there are three disqualifications in the list, then the Transfer Acknowledgement should be signed by the 4th relay in the list, since the previous three have been disqualified and that means that the 4th currently has authority for that deposit address.

Most transfers are acknowledged within a few seconds, and can be considered irreversible a few seconds later. Flex uses a wait time of 8 seconds. This is necessary due to the small chance that the authorized relay is itself performing an attack. This would require, in addition, that the authorized relay is the author of the two conflicting Deposit Transfers, since no node would pass along a conflicting Deposit Transfer after receiving a Transfer Acknowledgement.

If such an attack does occur, it is necessary to repeat the wait with the second relay in the list, and the third if necessary and so on.

## X. Credits, Batches and Mined Credits

Flex uses a subtly different type of structure to relate transactions, mining and expenditures. While other cryptocurrencies, following Bitcoin's example, group transactions together into blocks, Flex groups credits together into batches.

### I. Credits

A *credit* is simply a data structure which contains a number, indicating an amount of FLX, and a public key or the hash of a public key, which has the right to spend that money. It's comparable to the concept of a transaction output in bitcoin, although not all credits are outputs of transactions. *Credits* is also the term used (instead of coins) to refer to actual quantities of FLX.

There are three sources of credits: Transaction outputs, mining, and relay fees. Relay fees aren't paid by anybody; each relay who hasn't been disqualified for not responding or misbehavior receives a fee with no corresponding transaction approximately once per day.

*Mined credits* are also not transaction outputs. They're the result of mining, and, in addition to the key data and amount, they also contain information which encodes the state of the network. So while a regular credit looks like $(K, a)$, where $K$ is a public key and $a$ is an amount, a mined credit looks like $(K, a, s)$ where $s$ contains enough data to represent the state of the network[15]. We can, of course, drop the information about the state of the network and recover a simple credit from any mined credit.

### II. Batches

A *credit batch* is the closest thing in Flex to the concept of a bitcoin *block*. It's a collection of credits, grouped together and assigned numbers starting at a specified offset.

For example:

| Position | Amount | Public Key (Hash) |
|---|---|---|
| 47 | 100 | `21f9f7···` |
| 48 | 10 | `addcbd···` |
| 49 | 1 | `0f6f5f···` |

When mining, the miner collects recent credits (e.g. from transaction outputs) into a batch, with an offset determined by the number of credits encoded so far. In the example of the batch above, we presume that 46 credits have been encoded in previous batches, and so the newly added credits are assigned numbers 47, 48 and 49.

Using a symmetric hash function of two variables: $H(x, y) = H(y, x)$, the credits in the credit batch, together with their positions, are taken to be the leaves of a hash tree[16], and the root, $b$, of this tree is called the *batch root*. This is a 20-byte hash which is part of the informa-

[15] See the Flex Protocol document for an exhaustive specification of the information encoded in a mined credit

[16] or so-called Merkle tree

tion contained in the variable $s$, described in the previous section.

Because the batch root, $b$, is the root of a hash tree whose leaves are the credits with positions, there is a branch of the tree which connects each position plus credit, $(p, K, a)$, to the root. This branch is a sequence of hashes, $h_1, h_2, \cdots, h_n$, such that $H(\cdots H((p, K, a), h_1), \cdots, h_n) = b$, meaning that if we use $H$ to combine $(p, K, a)$ and $h_1$ into a 20-byte hash[17], and then combine the result of that with $h_2$, and then combine the result of that with $h_3$ and so on up to $h_n$, then the final result will equal $b$.

## III. The CreditInBatch

The purpose of this construction is to allow most users to avoid downloading all previous credits; it's enough to know the batch roots. If a user knows the batch root for a particular batch, but doesn't know the individual credits inside it, another user can present a credit plus position, $(p, K, a)$, along with the branch. The user can then perform the hashing sequence described above and, having found that the result is $T$, which matches the known batch root, the user can conclude confidently that the proffered credit at position $p$ is really inside the batch.

The data structure which, in addition to the credit $(K, a)$, also contains the position, $p$, and the branch, $h_1, \cdots, h_n$, is called a CreditInBatch.

These are the objects which we keep in our Flex wallets. Inputs to transactions are always CreditInBatch objects. These are credits that have been encoded into the state of the network through mining, and have been assigned a determinate position.

Transaction outputs are merely Credit objects, and it's not until they get encoded into CreditInBatch objects that they acquire a specific position and can be spent.

[17] That is, compute $H((p, K, a), h_1)$

## XI. Mining Credits

When a miner produces a mined credit, he or she gathers together the information, $s$, required to represent the state of the network, chooses a key, $K$, and specifies that the amount of the credit is equal to the subsidy for mining (namely 0.05 FLX).

Then the miner uses the hash of the mined credit $(K, a, s)$ as the seed for a proof of work, and starts to compute this proof of work. If the miner completes the proof of work before any other miners, the mined credit, proof of work and the list of messages[18] which were processed to bring the network from the previously encoded state to $s$, are bundled together into a Mined Credit Message which is then broadcast throughout the network.

The credit, $(K, a)$, payable to the miner then becomes a "loose credit", which the next miner will gather into a batch.

This is slightly different to the way that cryptocurrency mining usually works, wherein the miner's own block contains a transaction paying the mined coins to the miner. In Flex, the miner receives the credits one batch later than the one he or she mines.

## XII. Proof of Memory Occupation

Flex uses a custom mining algorithm called Twist to force the miners to occupy 4Gb of RAM while mining. The full details of the algorithm are available in the Flex Protocol document.

In brief, the input is used as a seed, $x_0$, to generate random numbers, and these numbers fill 4Gb of memory. The same input is then used to seed a different random number generator, and produce a sequence of numbers, $x_1, x_2, \cdots$.

Each number generated is interpreted as a location, $L$, in the 4Gb of memory (by taking $x_i$ mod $N$, where $N$ is the number of locations in 4Gb). Some of the bytes in $x_i$ are combined with some of the bytes at the location $L$ in

[18] It's actually a list of hashes of messages.

RAM, and the result is fed into a hash function. If the data that comes out of the hash function, interpreted as a number, is below a specified target value, then the number of steps, $i$, taken is recorded and the proof of work is over.

Checking a proof of work as simple as this: "Use $x_0$ as a seed and you'll find a hash below the target value after exactly $i$ steps", requires repeating the entire proof of work.

To make it easier to check, there is a second, more lax threshold used, called the *link threshold*. If the hash achieved at a given step is below the link threshold but above the target value, the number $i$ is recorded, the bytes, $V$, at $L$ are recorded, and then $i$ is reset to zero and $V$ is used to seed the random number generator which is generating the sequence of $x$ values.

This has the effect of breaking the proof of work, $(i, x_0)$ into several mini-proofs, $(i_0, x_0), (i_1, V_1), (i_2, V_2), \cdots, (i_n, V_n)$, where each mini-proof (called a link) can be checked by seeding the random number generator with $V_m$ and iterating for $i_m$ hashes. If the link threshold is not reached at step $i_m$, or if it's reached before that iteration, it means the proof is invalid.

The scheme used to fill RAM with random numbers is similarly broken into independently-generatable chunks. This allows checkers to use a smaller piece of RAM than the full 4Gb when checking a proof. If the location $L$ at each iteration falls within the area that the checker is checking, then the hash for that iteration is computed and compared to the link threshold. If not, the checker proceeds to the next iteration.

Finally, for a very quick check, the input to the hash which produces the final sub-target output value is also stored with the proof of work, so that a checker can quickly check that enough work to find a sufficiently low hash was performed, although further checking of individual links in the proof is required to prove that the worker did in fact allocate 4Gb of RAM during the work.

## XIII. Relays Again

If a miner, having mined a credit, intends to stay online and keep mining, then the miner can send a Relay Join message, specifying a public key which the miner (now a relay) intends to use for communication, as well as parts of the corresponding private key, to be distributed to executors. If the relay's successor has already joined, the Relay Join message also contains the successor's part of the private key; otherwise the relay waits until the successor joins before sending that.

Because traders, depositors and other relays must always agree about which relays were chosen to handle a specific set of secrets, information about the relays known to the network, about which relays have been disqualified, and about which relay is the successor of which other one, all need to be encoded into the state $s$ of the network which is included with each mined credit.

To accomplish this, a *relay state* is constructed, which contains a mapping from relays to their corresponding numbers as well as a record of the hashes of the relays' join messages. The relay state also contains lists of Succession messages received for each relay[19] and also hashes of any message (e.g. a complaint or refutation) which disqualifies the relay.

The relay state is serialized as a string of bytes and the 20-byte hash of this serialized state is included as part of $s$.

When a new Mined Credit Message arrives, one of the checks performed is to retrieve the relay state from the previous mined credit and apply all the messages referenced in the Mined Credit Message to the previous batch's relay state[20] and then see if the resulting relay state's hash matches the 20-byte hash contained in $s$.

[19] These are sent by executors on the occasion of a relay's misbehavior or nonresponsiveness. If 5 are sent, it means that the successor has inherited the relay's duties.

[20] Applying a Relay Join message adds a relay to the state; applying enough Succession messages marks the corresponding relay as defunct etc.

## XIV. The Calendar

The bitcoin blockchain is currently about 50 gigabytes in size. To achieve a quick start-up time, Flex needs a mechanism which permits users to start using it without downloading the entire history of every trade and transaction.

Flex uses a second difficulty threshold in order to reduce the amount of data which needs to be downloaded by new users to synchronize with the rest of the network. This difficulty is called the *diurnal difficulty*, and it's adjusted so that one batch per day, on average, has a proof of work which passes the threshold.

The period between these batches is called a *diurn*, to indicate that it's like a day, but isn't exactly a day. The diurnal difficulty is about 1,440 times larger than the difficulty required to mine a credit. Each time a batch appears with a proof of work which passes the threshold, the current diurn comes to an end and a new one begins.

When this occurs, the mined credit is called a *calend*. These calends, which appear once a day, are kept in a compact data structure called a *calendar*. It's so called because there's one entry in it per day on average and the state of the network at the end of each day is encoded in the corresponding calend.

A new user, instead of downloading all previous mined credits, downloads the calendar. The calendar has approximately 1,440 fewer entries than the entire history of mined credits, but each entry in the calendar proves approximately 1,440 times as much work as a single mined credit[21].

The total work in the calends in the calendar is equal to the sum of the diurnal difficulties, just as the total work in a regular blockchain is equal to the sum of the block difficulties in a regular cryptocurrency. However, because the arrival of calends is stochastic, the total work in the calendar up to the latest calend may be greater than or less than the total batch work done up to that point.

In practice, each mined credit encodes within $s$ the total amount of batch work[22] completed up to that point. If this amount is greater than the work in the calends plus the current diurnal difficulty[23] then that mined credit and its proof of work are stored along with the calendar as "top up work", so that, should this mined credit turn out to be a calend, the total work in the calendar, including top up work, will equal or exceed the total work reported by the mined credit.

This means that new users who receive a mined credit, which purports to be the tip of a long chain containing a certain amount of work, don't need to measure all that work. Instead, each new user checks that the work in the calendar plus the top up work exceeds the amount of work reported in the mined credit. This allows the user to accept the fact that there is as much work behind that mined credit as it claims.

The appearance of a calend is the occasion when relays get paid their fees (provided they haven't been disqualified). By paying relays once a day instead of once per trade or transfer, the number of credits paid to relays is limited to 600 per day, allowing the number of trades and transactions per day to increase without introducing a danger that the system will be swamped with fees for relays.

### I. Diurn Roots and Diurn Branches

Between one calend and the next, we have a list of batches or mined credits which have accumulated during this diurn. Since we don't want new users to download these, but we

[21] Each calend has two difficulties and proves two different amounts of work. The fact that the proof of work's final hash was below the threshold needed for a *batch* proves that the worker successfully sought and found a hash small enough for that. The fact that it was below the threshold needed for a *calend* proves additional work only insofar as the worker was actually seeking a hash that small - i.e. only insofar as the diurnal difficulty is specified in advance.

[22] meaning work which is calculated as the sum of the difficulties which the miners needed to reach to encode each batch up to this point

[23] which is equal to the amount of work that the the calendar will have (excluding top-up work) when the current diurn ends.

do want new users to be able to determine the validity of CreditInBatch objects from these batches, we need a method for validating such credits using only the calends.

To accomplish this, each miner encodes within $s$, the state of the network, a *diurnal block root*, $d$, which is the root of a hash tree whose leaves are the hashes of the mined credits which have appeared so far in the diurn. Each mined credit hash, $h$, is connected to $d$ via a branch (called a *diurn branch*), which is a list of 20-byte hashes, $g_1, \cdots, g_m$.

Furthermore, the mined credit hash, $h$, is constructed in a special way: the batch root is obtained, the hash of the rest of the mined credit data, $r$, is obtained (this hash is called the *branch bridge*), and then these are combined with the symmetric hash function $h = H(r, b) = H(b, r)$ used for constructing hash trees, to obtain $h$.

This means that, when a calend does arrive, it contains a diurnal block root $d$, and each mined credit since the beginning of the diurn is linked to $d$ via a diurn branch: $H(\cdots H(h, g_1), \cdots, g_m) = d$.

Since an individual credit is linked to the batch root, $b$, via a branch, and $b$ is linked to the credit hash, $h$, via the branch bridge: $h = H(r, b)$, and the credit hash, $h$ is linked via the diurn branch to the diurnal block root which is recorded in the calend, there is in effect a long branch:

$$h_1, \cdots, h_n, r, g_1, \cdots g_m$$

connecting an individual credit from a diurn all the way to the diurnal block root in the calend.

This allows users who possess only the calendar to validate CreditInBatch objects from any previous diurn, provided the diurn branch in included. The diurn branch is added to the CreditInBatch object before it is used in transactions.

## XV. The Spent Chain

With the assistance of a calendar, a user can confirm that a credit plus position, $(p, K, a)$, offered by a spender who provides the batch branch and diurn branch, is in fact a valid credit belonging to a diurn whose calend is present in the calendar.

Before accepting the CreditInBatch object as a transaction input, each user needs to be able to confirm that the specific credit wasn't already spent.

To accomplish this, each user keeps, along with the calendar, a string of 1's and 0's indicating which credits have already been spent. This is called the *spent chain* and it contains a 1 in the $n^{\text{th}}$ place if and only if the $n^{\text{th}}$ credit (i.e. the credit $(p, K, a)$ with $p = n$) has been spent.

The rules for updating the spent chain are easy: When a new transaction arrives, we add a zero at the end of the spent chain for each output, and flip an earlier zero in the chain to a 1 for each input. Credits that aren't transaction outputs (mined credits and fees) also result in zeroes added to the end of the chain.

The hash of the spent chain is included in the state, $s$, of the network encoded into each mined credit.

New users will need to download the latest spent chain as well as the latest relay state, calendar and all of the batches since the last calend.

The spent chain grows indefinitely as more transactions occur and mined credits are found. It is, however, significantly smaller than a blockchain, containing a single bit for each credit rather than dozens of bytes. It's also possible that the spent chain could be compressed to a small fraction of its uncompressed size, unlike the blockchain which is mostly hashes.

## XVI. Open Questions

This field of research is sufficiently new that most questions that could be asked are open, but here we'll list just a few of the questions most relevant to Flex.

- Should the mining reward change over time?

Bitcoin's finite supply leads many to conclude that its value will rise over time. How-

ever, the reward for miners diminishes over time and some day miners may rely on fees for most of their income. Credits in Flex are not intended to be a long-term store of value, and their value will be determined by their popularity as a means of exchange, so no halving of the production rate has been programmed and the supply of Flex credits will increase indefinitely.

No analysis has been done to establish that a fixed rate of production is optimal for any specific purpose.

- What should the ratio of the reward for mining to the fee or reward for correctly handling secrets be?

Currently, the reward for mining a batch is 0.05 credits, and the reward for not getting disqualified is 0.01 credits. These numbers were chosen rather arbitrarily. There might be an optimal ratio, or perhaps the optimal ratio depends on some parameter such as the number of recent complaints and refutations.

- Are refutations necessary?

In the scheme of complaints and refutations used in Flex, refutations were used because there's no clear way, other than revealing a private key, that a recipient can prove to a third party that the correct secret number (corresponding to a particular elliptic curve point) was *not* sent. The refutation was a proof that the correct secret was indeed sent. If there's a way for the complaint to include a proof that the secret sent was bad, rather than merely an affirmation, the refutations can be dispensed with.

## XVII. Scaling

Although Flex has some features that scale well, such as the calendar, it is quite plausible that the total bandwidth taken up by relaying all the secrets involved in all trades, transactions, deposits and so on could exceed the capabilities of a typical household in many parts of the world.

One option would be simply to have more than one Flex network running. If one network handled bitcoins and litecoins and another handled other cryptocurrencies then they could coexist without sharing each other's channels of communication.

A problem with that approach is that it splits the mining power into two parts and decreases the network security. This could possibly be solved by developing a type of merged mining in which miners could mine blocks for multiple networks at once without traffic from one network taking up bandwidth on another.

Trade between the multiple Flex networks would be very easy, since each network would be able to handle the credits of the foreign Flex network just as easily as any other cryptocurrencies. The credits in one network wouldn't necessarily have the same value as credits from the other network, although further research may make this possible.

### I. Transferring Deposits Between Flex Networks

With a few minor modifications to the code, it's possible for somebody connected to two different Flex networks to transfer deposits between them. The Withdrawal Request Message described in §IX.I contains a *recipient key*, which is not necessarily the same as the key used to sign the withdrawal request. This means that you can ask the relays to tell somebody who might not be you what the secret parts of the deposit address's private key are.

The public key of the deposit address on the destination network can be used as the recipient key for the withdrawal. Whoever learns the private key of the destination deposit address will be able to read the secret parts of the original deposit address key, and can then import the key into his or her wallet. The double-checkers can stand guard to ensure that the correct secrets are sent during the withdrawal.

The withdrawal messages could be published out of band, or if they can be validated, they could be broadcast on the destination network, so that everybody can see the connection

between the destination deposit address and the coins at the original deposit address.

With such an arrangement, the number of transactions that can be processed per second, in parallel but interoperable networks, is effectively unlimited.

## XVIII. Outlook

With the increasing interest in bitcoin and cryptocurrencies in general, the importance of moving transactions off-chain is growing. Passing many parts of secrets instead of spending currencies is one possible way in which cryptocurrency transmission could be effected on a large scale without filling up blockchains. The system of deposit transfers used in Flex allows fast, irreversible tranfers within seconds, using two short messages each, and no permanent record is kept in a blockchain[24].

Decentralized exchanges, with no single trusted party, have long[25] been a goal of cryptocurrency development. Flex is one implementation which demonstrates that splitting, passing and recombining secrets is a viable and efficient way of trading without a trusted exchange.

These are two applications of secret passing. No doubt there will be many others.

## XIX. Acknowledgements

Cryptocurrencies themselves would not exist were it not for the groundbreaking work of Adam Back[2], Nick Szabo[3], Wei Dai[4] and Satoshi Nakamoto[5].

In addition to important contributions to cryptocurrency theory and practice which have informed the ideas implemented in Flex, thanks are due to Sergio Demian Lerner[6] and Juliano Rizzo for identifying vulnerabilities in the implementation.

Adi Shamir[7] developed the *k*-of-*n* secret sharing scheme which Flex's is built upon, while Whitfield Diffie and Martin Hellman's[8] procedure for communicating via a shared secret was the precursor of Flex's cipher.

Colin Percival's scrypt algorithm[9] was the original inspiration for the Twist proof of work, and the pioneering contributions to the theory of memory-hard proofs of work from John Tromp[11], Daniel Larimer[10], and more recently, Alex Biryukov and Dmitry Knovratovich[12] have significantly advanced the field.

## References

[1] TLS Notary (2014). PageSigner - One-click website auditing *https://tlsnotary.org/pagesigner.html*.

[2] Adam Back (1997). Hashcash - A Denial of Service Counter-Measure *http://www.hashcash.org/papers/hashcash.pdf*.

[3] Nick Szabo (1998). Bit gold *http://unenumerated.blogspot.de/2005/12/bit-gold.html*.

[4] Wei Dai (1998). B-Money *http://www.weidai.com/bmoney.txt*.

[5] Satoshi Nakamoto (2008). Bitcoin: A Peer-to-Peer Electronic Cash System *https://bitcoin.org/bitcoin.pdf*.

[6] Sergio Demian Lerner (2013). Strict memory hard hash functions *https://bitslog.files.wordpress.com/2013/12/memohash-v0-3.pdf*.

[7] Adi Shamir (1979). How to share a secret *http://dl.acm.org/citation.cfm?id=359176*.

[8] Whitfield Diffie, Martin E. Hellman (1976). New Directions in Cryptography *https://ee.stanford.edu/ hellman/publications/24.pdf*.

[9] Colin Percival (2009). Stronger Key Derivation Via Sequential Memory-Hard Functions *https://www.tarsnap.com/scrypt/scrypt.pdf*.

[24] Although somebody recording all traffic on the Flex network would be able to deduce which public keys had exchanged coins.

[25] at least, long compared to the median lifetime of cryptocurrencies

[10] Daniel Larimer (2013). Momentum - A Memory-Hard Proof-of-Work via finding Birthday Collisions. *http://www.hashcash.org/papers/momentum.pdf*.

[11] John Tromp (2014). Cuckoo Cycle: a memory bound graph-theoretic proof-of-work *https://eprint.iacr.org/2014/059.pdf*.

[12] Alex Biryukov, Dmitry Khovratovich (2015). Asymmetric proof-of-work based on the Generalized Birthday problem *http://eprint.iacr.org/2015/946.pdf*.