Breakout Chain White Paper

Abstract

Breakout Chain is a decentralized blockchain ledger that carries several currencies, which we term a multicurrency blockchain. Each of these currencies is a first class currency in that each is transferrable using the Bitcoin-like input/output mechanism. The principal currency is Breakout Coin, but the blockchain is secured by Proof-of-Work (PoW) by mining Sister Coin and Proof-of-Stake (PoS) by staking Breakout Stake. It is the first blockchain system where interest earning capacity has been separated from the principal. It is also the first blockchain system where PoS and PoW produce different coins.

Breakout Chain has a number of other features such as delegated fees, fee scavenging, nonmalleable transactions, and atomic currencies. Each described herein.

In addition to these existing features, Breakout Chain plans a Turing complete smart contract system that is amenable to pegged sidechains, making Breakout Chain fully scalable. Breakout Chain smart contracts will have the property of directed acyclic graphs by use of transaction control points that regulate logical recursion.

The combination of existing and planned features will allow for a breadth of applications, some of which are outlined herein.

Background

Bitcoin is Decentralized Cash

The definition of ***decentralized cash*** is a money system that lacks a central authority to approve transactions. Arguably the first fully viable decentralized cash is ***Bitcoin*** (ticker symbol BTC), launched in 2009 by one or several individuals working under the potentially pseudonymous moniker Satoshi Nakamoto [1]. The Bitcoin protocol serves as a model for every decentralized money system currently in use. For this reason, in lieu of saying “Breakout Chain”, this document makes many references to “Bitcoin” as a shorthand for “the Bitcoin protocol upon which Breakout Coin is based”. This usage is meant not only to save space, but to give credit to the Bitcoin creators where appropriate.

Ethereum is a Decentralized Computing Platform

Ethereum is a highly experimental decentralized computing platform invented by Vitalik Buterin [2]. Ethereum extends Bitcoin by offering ***Turing complete*** scripting (scripts are simply short computer programs). Turing completeness is the ability to simulate a ***Turing machine***, which is a hypothetical machine consisting of an instruction processor and an infinitely long memory tape that can seek forwards and backwards. This esoteric sounding definition can be thought of (perhaps imprecisely) as

describing a machine that can execute any logic accessible to computers. Turing completeness requires the ability for a computer program to loop one or more times through the same set of instructions, in a process known as ***unbound recursion***.

Turing completeness endows Ethereum scripting with tremendous power to execute many kinds of contracts that represent fiducial relationships between people. These types of contracts are often called ***smart contracts*** [3] to reflect the fact that the computer program can execute the terms of the contract, presumably replacing human intelligence. Even ***decentralized autonomous organizations*** (DAOs) [4] – that work like companies – have been launched on the Ethereum platform. DAOs are also known as decentralized autonomous corporations (DACs) because it is possible for people to buy shares in these organizations, with ownership being represented on Ethereum.

With Ethereum's Turing complete power comes one critical burden known as the ***halting problem***, which is the general inability to determine in advance whether a computer program will complete in a finite amount of time. In other words, the only way to know if a Turing complete program will finish is by running the program. The halting problem means that the execution of programs in a Turing complete programming language must have controls in place that terminate execution even if the program itself has not completed. Ethereum uses its native currency, ETH, as a form of "gas" for this control. Each program is supplied with gas money and once the program runs out of gas, execution is terminated. If the program did not complete successfully, the gas money is forfeited as a fee and, except for payment of the fee, the state of the Ethereum computing platform remains unchanged, as if the computation had never happened.

Another challenge with Ethereum is that it may not be a viable system when the frequency of transactions and scripts increases beyond a threshold. This problem is known as the ***scalability problem*** and effects all decentralized cash systems. This problem is especially severe for Ethereum because each script affects global state of the system. This means that Ethereum scripts must have a ***strict ordering*** (script X must finish executing before script Y can start) so that the state of the system is known with certainty for the execution of every script.

Ethereum's strict ordering precludes the ability to run more than one script at the same time, a technique called ***concurrency***. To see how ordering precludes concurrency, consider what would happen if all of Ethereum's activity were split into two sets A and B, termed ***shards***. In this hypothetical example, the individuals who run shard A take with them half of the ETH money supply to be used for gas while the individuals who run shard B take the other half. Scripts from both shards are executed concurrently as long as (1) ***crosstalk*** between shards is prevented such that activities from shard A do not require input from shard B and vice versa, and (2) eventually the two shards fuse so that ETH money can be used as a payment system for the entire Ethereum platform.

While it is fairly trivial to prevent crosstalk, the real (and perhaps insoluble) problem is that it is impossible to fuse shards without members of each shard fully validating the other shard. The reason for validation is to ensure that the ETH that ends up in the fusion has not been spent as gas in either of shards. This need for validation destroys any benefits of concurrency.

Breakout Chain Proposes Decentralized Computing with Concurrency

Proposed Computing Platform

To address the scalability and halting problems faced by Ethereum, Breakout Chain proposes a Turing complete scripting engine that uses a different control mechanism from Ethereum. Instead of Ethereum's approach of allowing each script to run to completion before deciding whether it can modify the state of the computing platform, Breakout Chain will divide execution of contracts into segments. These segments are restricted from recursing except when users push execution through loops by submitting a transaction. These control points mean that scripts can be filtered for length and complexity without the need to execute them, allowing for greater scalability.

Although this mechanism may appear cumbersome at first sight, I will demonstrate through an example of crop insurance that control points are natural places at which actors in a contract would make transactions. To see this, imagine the recursion involved in a monthly subscription represented as a smart contract. Each loop through the contract would be initiated by a payment. In fact, because the contract mediates a benefit in exchange for payment, it is desirable to halt the execution of the contract until payment is made. This simple principle forms the foundation of the Breakout Chain computing platform, and allows the platform to have properties that make it amenable to scaling through concurrency.

For this reason, the proposed Breakout Chain protocol permits shards through the use of ***pegged side chains*** as described by Adam Back, *et al*. [5]. Pegged side chains are shards wherein money from the main shard (also called the ***main chain***) is locked and then apportioned to a separate shard (the ***side chain***). The money can be used in the side chain and until it is destroyed while concomitantly unlocking the money on the main chain. Pegged side chains require both the main chain and side chains to have properties that make them amenable to concurrency because the side chain must run concurrently with the main chain.

Existing Features and Innovations

Although Breakout Chain will not have the Turing complete scripting engine at launch, it already incorporates a number of new features that will be described in detail below. These features include

- ***Multicurrencies*** – multiple currencies carried on a single chain, each currency having a full set of features available to it, like multi-signature transactions, locked transaction times, and sophisticated scripting
- ***Multicurrency Hybrid PoW/PoS*** – a unique security mechanism where currency A is rewarded to secure a chain using *Proof-of-Work* (PoW, described below) and currency B is rewarded to secure a chain using *Proof-of-Stake* (PoS, also described below)
- ***Currency Interrelationships*** – Proof-of-Stake for currency C is rewarded with currency D

- ***Breakout Gravity Wave*** – a novel difficulty algorithm derived from *Dark Gravity Wave* that throttles ***block*** creation (a block is a group of transactions bundled as a single unit of data, similar to a ledger sheet)
- ***Atomic Currencies*** – currencies that are indivisible beyond whole units, making it possible to represent real world fungible assets that can't be divided, like frequent flyer miles or loyalty points
- ***Delegated Fees*** – the ability to send currency A but pay for the transaction in currency B
- ***Fee Scavenging*** – the ability for transaction fee collection to be deferred until a Proof-of-Work claims the fee
- ***Nonmalleable Transactions*** – transactions protected against changes to their transaction IDs, addressing a current vulnerability in bitcoin
- ***Colored Coins*** – currencies with only one unit that is indivisible, a proof of concept for which has been incorporated into Breakout Chain in the form of a deck of cards
- ***Exchange Friendliness*** – exchanges can add any currency supported by Breakout Chain in several seconds by modifying only one line in the configuration file

Several other features derived from other decentralized currencies have also been incorporated into Breakout Chain, including a full-featured graphical wallet for several platforms, free-form transaction metadata, stealth addresses, and encrypted transactions comments that can only be read by sender and receiver.

Concepts

Rather than present a complete history of electronic money systems as background, the first part of this white paper reviews several essential aspects of existing payment systems. Knowledge of these topics is important for understanding novel aspects (both existing and proposed) of the Breakout Chain protocol.

Decentralized Ledgers

The essence of a ***ledger*** is simply a table of account credits and debits. Ledgers have starting balances, then current balances can be tallied simply by adding all ***credits*** (receipts) and subtracting all ***debits*** (payments). An example is the following table for two accounts, Alice and Bob:

Table 1: Example of a Simple Ledger Sheet

	Alice			Bob		
Memo	**Credit**	**Debit**	**Balance**	**Credit**	**Debit**	**Balance**
Starting			$0			$0
Alice Deposit	$100		$100			$0
Bob Deposit			$100	$2		$2
Alice pays Bob		$10	$90	$10		$12
Bob pays Alice	$11		$101		$11	$1

This example demonstrates several aspects of a typical ledger. First, for any ***internal transfer***, the sum of all credits is equal to the sum of all debits. For example, when Alice pays Bob, the debits are 10 and the credits are 10. This means that internally, ledgers adhere to the principle of ***conservation of money***. Notice that all entries are not internal transfers. One entry, Alice's initial deposit, was an external transfer that added money to the ledger.

Each line of a ledger can have a pictorial representation, where the ledger itself could be considered a middleman, or ***payment processor***, that mediates exchanges. Figure 1 depicts the second transaction in the ledger above, where Alice pays Bob $10.00.

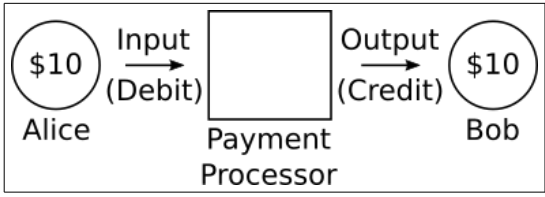

Figure 1. Alice sends $10 to Bob. This transaction can be imagined as one where a Alice creates an input (debit) from her account and an output (credit) to Bob's account.

The reason for placing the payment processor as a mediator of the transaction is to introduce a visual framework for the concept of ***inputs*** and ***outputs***. Namely, the payment processor is placed visually in the middle to establish a perspective for what is “in” and what is “out”. Roughly, an input can be imagined as a debit from an account while an output can be imagined as a credit. In this example, Alice created an input of $10 that got debited from her account. By transferring the money to Bob, she also created an output of $10 that got credited to Bob's account.

In this framework, Alice had to perform some operations to create the input and output, so we could say that the process she followed adhered to a particular ***input/output mechanism***. This terminology communicates that some computational machine does Alice's bidding, and within this machine is the *mechanism* to process the *inputs* and *outputs*. This computational machine is given the name “payment processor”.

Traditional ledgers and some distributed ledgers like Ethereum, Nxt [7], Counterparty [8], and

BitShares [9] explicitly pool all of an account's funds to a single balance at every entry, similar to what is depicted in Table 1. For the sake of convenience, these type of ledgers will be termed herein as ***state ledgers***. In state ledgers, an account's funds are represented as a single number, which is the balance. In a sense, an account's balance is change from a previous transaction that used all of the previous balance as an input. Figure 2 depicts change represented as an output. In the context of the ledger in Table 1, this change output becomes Alice's balance.

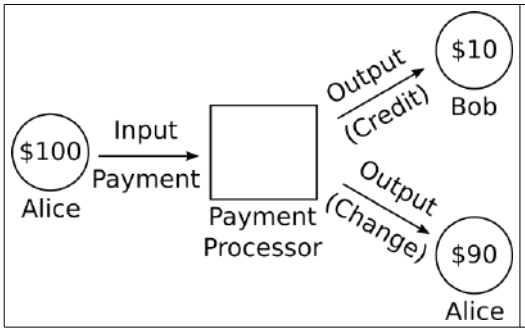

Figure 2. A state ledger implicitly makes change. In this transaction, Alice begins with a $100 input (her previous balance) and sends a $10 output to Bob, which is a credit to his account. The transaction is balanced with an implicit change output that in this case becomes Alice's new balance.

Having established the concept of inputs and outputs and their relation to ledger entries, it is now possible to understand the specific needs of a ***decentralized ledger***, such as that represented by Bitcoin. First, what is a decentralized ledger? A decentralized ledger is one where copies of it are maintained by many different parties separated by distance. Additionally, the different parties potentially have conflicting interests. The integrity of a decentralized ledger depends on each copy of the ledger being updated in exactly the same way.

An important requirement for a decentralized ledger is that updates to it must be synchronized. That is, not only must the copies be updated in exactly the same way, these updates must occur close enough in time such that any copy can be relied upon to faithfully report an account's balance at any moment.

A second requirement of a decentralized ledger is that account holders need to ***prove*** that they are authorized to spend funds, or more precisely, prove that they own the account to which funds were sent. With a centralized ledger (like a bank uses), this proof might take the form of signing a check, using a debit card with pin, or presenting an ID at a teller.

With decentralized ledgers, a different form of proof, called a ***cryptographic signature***, is required. The cryptographic signature is produced by a computer program that has access to the spending account's secret information that identifies the account uniquely. To spend an input, the input is signed via the cryptographic signature.

A cryptographic signature can be quite complicated as it may need to provide a proof in multiple parts. Examples might be that an input requires that two or more parties must both sign it (***multi-signatures***), or that the spender must prove knowledge of some other information aside from the account's identifying secret. The technical details of signatures and multi-part proofs are quite elaborate and are beyond the scope of this white paper, although it helps if the reader appreciates the power behind

complex spending requirements like multi-signatures. A typical use of multi-signatures, for example, is a 2-of-2 multi-signature that requires two different people to approve spending funds.

The combined requirement of synchrony and potentially complicated proofs of spending authority (the cryptographic signature) imposes constraints on decentralized ledgers. These constraints mean that is often not most efficient to combine of all of an account's inputs (credits) into a single spendable unit that most people think of as a balance. Using multi-signatures as an example, it is obvious that if two different inputs require two different sets of signatures, they can not be spent as a combined unit. In other words, the first input may need multi-signature authorization and the second input may only need single-signature authorization. Therefore, it is not sensible to combine them into a single balance upon receipt. Instead, a more efficient approach is to wait to ***combine inputs*** until they are spent, signing only those needed at the time.

In short, inputs are treated as discrete units that are signed independently by the spender. In cases where all of a spender's inputs are smaller than a given payment, the spend will combine inputs to produce the payment. Figure 3 illustrates combined inputs.

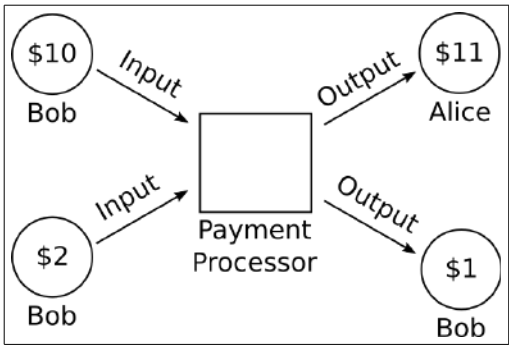

Figure 3. Inputs are combined to cover a spend larger than any single input. In this transaction, Bob pays Alice $11. Bob only has inputs of $10 and $2 available, so he combines them and then makes an $11 output to Alice and a $1 output to himself as change.

To distinguish state ledgers from ledgers that treat inputs as discrete units, this white paper will refer to the latter as **UTXO ledgers**. The acryonym UTXO stands for “unspent transaction output”, a phrase that describes a transaction output before it referenced by an input that is signed and spent.

State ledgers are fundamentally different from UTXO ledgers in that state ledgers require strict ordering of transactions while UTXO ledgers do not. The reason state ledgers require a strict ordering of transactions is because it is impossible for an account to have a negative balance. For example, assume that Alice has a balance of $20 and receives a deposit of $10 then spends $25. This set of transactions would leave her with $5. But imagine that a transaction processor receives the spend first, meaning her account would have a balance of $(5), invalidating the spend. If Alice’s account is on a decentralized ledger, she could walk away from her negative balance without any repercussions.

UTXO ledgers, on the other hand, are ordered by the input/output mechanism. In the example above, Alice might receive output A as a deposit and then spend output B. Output B points to output A as one of its inputs. If a network participant receives notification of Alice’s spend with output B, the participant will know to wait some reasonable time for output A because it has been referenced by output B.

Fees

Processing payments requires computational resources to validate and maintain a complete history of transactions, which means that it is desirable for users to pay fees. In addition to compensating the individuals who provide payment processing resources, fees ensure that users do not abuse those resources by creating too many transactions for the payment processor to handle. Although centralized payment processors like banks are able to automatically deduct a service fee each month from user accounts, decentralized payment processors can not deduct automatic payments. The reason is that account holders must specifically sign inputs to spend them and any payment transaction, like a fee, is a spend. Thus, fees paid using a decentralized ledger are taken for every transaction (even though these fees are generally very small). Figure 4 shows a transaction (**not** appearing in the ledger in Table 1) with a fee.

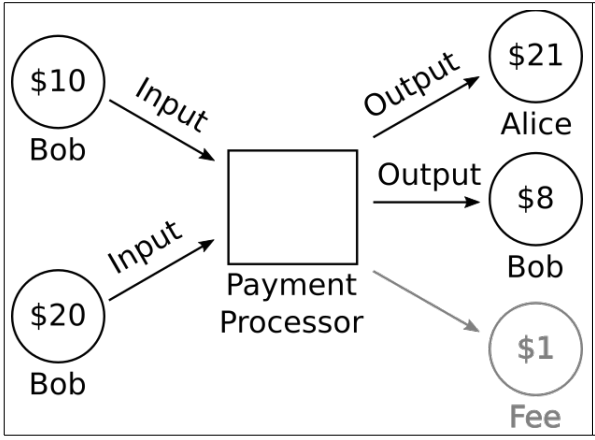

Figure 4. Fees are paid by leaving the appropriate amount of their inputs unclaimed. In this transaction, Bob will pay Alice $21. Bob only has inputs of $10 and $20, so he combines them and creates an output of $21 to Alice. $9 is left over, but Bob needs to pay a $1 fee, so he makes an output of $8 to himself. He does not specify where the $1 fee goes (indicated by gray). Instead, the payment processor will see that the $1 is unclaimed and claim it as a fee. The output transaction for the fee, therefore, is created by the processor and not by the spender (Bob).

Figure 4 illustrates that a decentralized ledger conserves money in that any (generally purposefully) unclaimed funds from a transaction may be claimed by an agent of the payment processor.

The Blockchain

The structure of a traditional ledger is presented in Table 1. The layout is intuitive. Each account has a debit column, a credit column, and a balance column. The ledger grows by one row with each new transaction. Traditional ledgers are rectified upon each internal transfer by ensuring that all the debits equal all the credits for the transaction.

Because of the potential asynchrony that accompanies decentralization, Bitcoin was designed with a different structure from traditional ledgers. The Bitcoin ledger is not continuous series of rows. Instead, it is divided into sets of transactions called ***blocks***. Each block can hold many transactions wherein each transaction is composed of inputs and outputs. The first transaction of a block generally includes a ***block reward***, a special output that increases the money supply and is rewarded to the individual who processed the block of transactions. The first transaction can also include fees if any are claimed by the processing individual. After this first transaction, called the ***coinbase transaction***, are transfers authored by account holders. These are the types of transfers depicted in Figures 1-4 above.

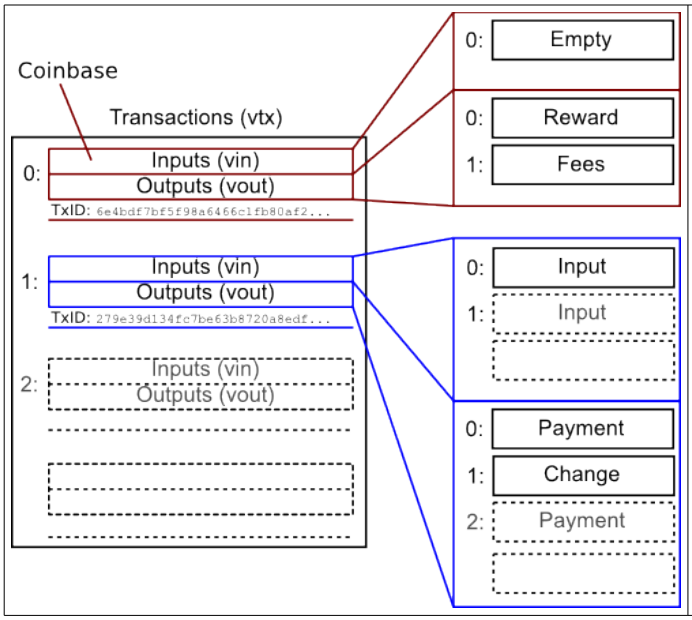

Figure 5. Blocks contain an array of transactions, called *vtx*. The first transaction in a block is called the coinbase transaction. Every transaction holds an array of inputs called *vin* and an array of outputs called *vout*. Notice that fees are paid in non-coinbase transactions but collected in the coinbase transaction as a fee output. Also, the coinbase transaction creates new coins, so does not have any inputs. Technically speaking, the input array is filled with one empty input as a placeholder to indicate that the coinbase transaction is correctly formed. Each transaction has record of its own identifier, called a transaction ID, commonly abbreviated "TxID".

Figure 5 illustrates a couple of important points. First each transaction stores a record of its own identifier called a ***transaction ID*** (TxID). Each TxID is unique. Second both the inputs and outputs for each transaction are numbered by their index within the array (idiosyncratically beginning with 0). Because TxIDs are unique to the blockchain and outputs have unique array indices, every output can be identified by a combination of its TxID and array index.

In addition to an array of transactions, blocks have other information, one of which is a very long number, called a ***block hash***, that identifies the block. Block hashes are expressed almost exclusively as hexadecimal numbers, meaning the number system has 16 digits from 0 to 9 then a to f, instead of just 0 to 9. An example block hash is:

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Although this hash may not look like a number, it is simply hexadecimal for the big number

```
106289448692185620840501435194445495803894645914546740193455560 79
```

Block hashes will often have several leading zeros, which is simply a formatting convention to make them exactly 64 characters long.

Each block is not only identified by a block hash, but also contains a record of the hash of the block that came immediately previous. This record is often referred to as a pointer to the previous block. These pointers link blocks together into a chain, giving motivation for the name "blockchain".

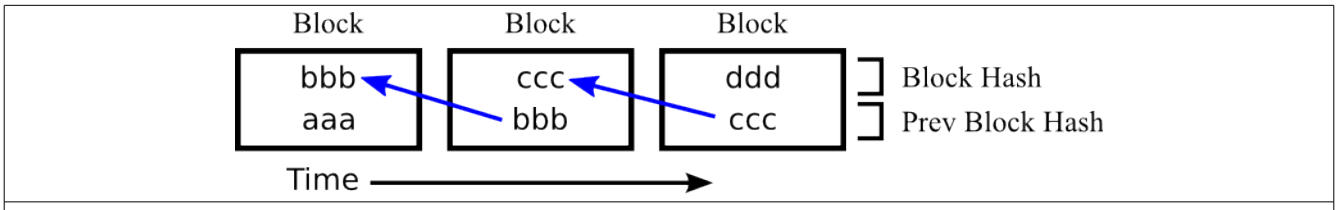

Figure 6. The blockchain is a series of blocks of transaction linked by pointers to the previous block hash. The strings “aaa”, “bbb”, “ccc”, “ddd” represent block hashes. Each block is identified by and contains a record of its own hash (“Block Hash”). Each block also contains the hash of the previous block (“Prev Block Hash”), that serves to link blocks into a chain.

Inputs and Outputs

At the heart of blockchain transactions are inputs and outputs. Outputs are simple data structures that have three parts: (1) a destination with spending requirements, (2) the amount to transfer to the recipient.

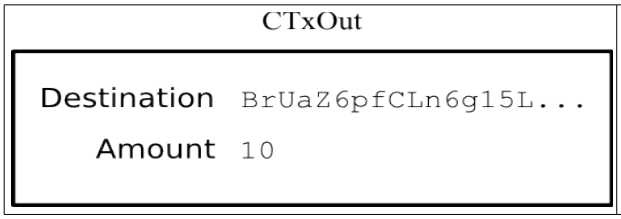

Figure 7. An output is a data structure with records only for the destination and an amount to send. The destination is typically simply an address, but could specify other spending requirements. An output is referred to as “CTxOut” in the computer code.

Inputs are slightly more complicated because they must contain proof that the spender has the right to spend the funds. This proof is generally called a ***signature***. In addition to this proof, an input must reference the previous output that is being spent. This reference takes the form of a TxID and an array index. Figure 8 illustrates how an input references the previous output.

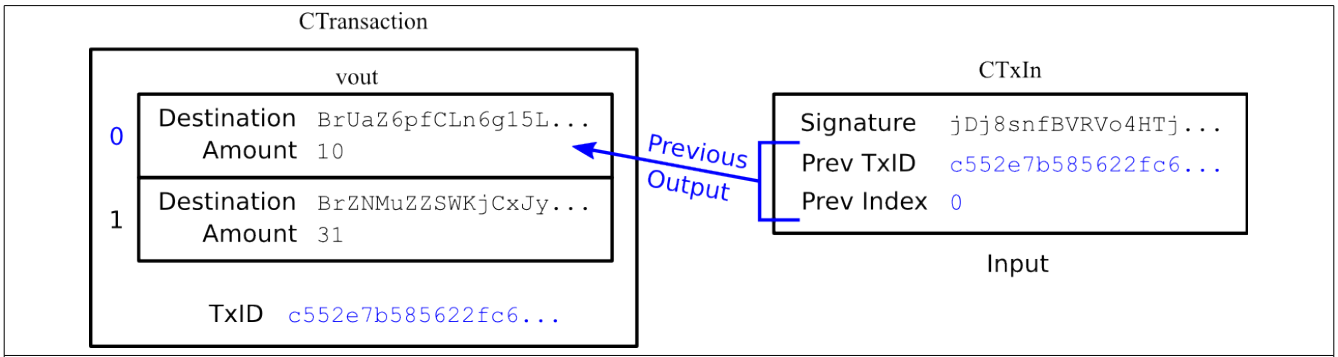

Figure 8. An input is composed of a signature (proof of authority to spend funds) and a two-part reference to the previous output that the input spends. In this example the previous output belongs to the transaction with a TxID starting “`c552e7b58...`” (blue) and having array index 0 (blue) in vout. The full data structure of the transaction is not shown. For example, all transactions will have an array of inputs called “vin”.

Blockchain Security

The Cryptographic Hash

Central to blockchain security is the ***cryptographic hash***, or “hash” for short. A hash is produced by a ***function*** (a block of computer code in practice) that takes a ***string*** (sequence of characters) as input and returns a very large unpredictable number as an output. In this context, “unpredictable” means that the only way to know the hash of a given string is to pass this string to the hash function and calculate the result. The cryptographic hash used extensively for decentralized ledgers is known as SHA256d.

To get a feel for what cryptographic hash functions do, consider the quote “*I believe that banking institutions are more dangerous to our liberties than standing armies*” by Thomas Jefferson, one of America's founding fathers, author of the Declaration of Independence, third U.S. President, and probably America's most revered patriot. The SHA256d hash (expressed as a hexadecimal number) of Jefferson's phrase is:

```
eaeaebf2f52b0ae9235ca3709bf2accd1acc15ff295762a35997e28e7ac35773
```

Aside from unpredictability, cryptographic hash functions have two other important properties. The first property is that it is practically impossible (within the limits of all resources available to humanity) to find two strings that will produce the same hash. The second property is that it is impossible to ascertain the string that produced the hash with any confidence. In other words, there is no function that could take a hash and return a string that would produce that hash without brute force trial and error. To illustrate this latter property, imagine Jefferson's quote if he had ended the sentence with “armies” and put a period after it. The string would be “*I believe that banking institutions are more dangerous to our liberties than standing armies.*” This means the period would go into the hash function as well. The SHA256d hash of this new phrase is:

```
72ed6a2be299ae41c09c681cca5d1093175b73d5e146a24bee2b4f3b638eeb97
```

Inspection of this latter hash reveals that it has no relation to the hash of the original phrase, despite the two phrases' differing by only one character, the sentence-ending period.

The properties of cryptographic hashes mean that they make very good building blocks for designing computationally difficult problems. For example, one problem might be to determine a string one would need to append to the end of Jefferson's quote to make the first hexadecimal digit of the SHA256d hash into a zero (“0”). Because of the cryptographic properties of SHA256d, the only way to work this problem is by trial and error. For the curious, the resulting string is “*I believe that banking institutions are more dangerous to our liberties than standing armies5*” (appending a “5” to the original quote), and the SHA256d hash is:

```
053e11f72f538d15e84ec15ba395c7b755cc85cfa6e4e3e29af446e04fc3ca9a
```

Because there are 16 possibilities for each hexadecimal digit (“0” to “9” plus “a” to “f”), converting each successive digit to a zero requires, on average, 16 times more effort than converting the previous digit. For example if it takes 1 unit of computational ***work***, on average, to convert the first digit to zero,

it will take 16 units of effort, on average, to convert the first two digits to zero. The resulting hash would be “00” with 62 hexadecimal digits following. To convert the first eight digits to zero (meaning a hash that starts “00000000”) would take, on average 268,435,456 units of effort. That is, it is over 268 million times harder to find a SHA256d hash with 8 zeros at its start than with just one zero at its start.

Although this example problem of converting starting digits to zeros seems pedantic, it is fundamentally the cryptographic “puzzle” that lies at the heart of blockchain security.

Proof-of-Work

What is the nature of a “transaction processor” for a decentralized ledger and what ensures that they act honestly? The activity of a transaction processor boils down to signing blocks that all members of the network accept into their copy of the blockchain.

Transaction processors demonstrate their authority by solving what is essentially the problem just described. In simple terms, the problem is to turn all of the block data (transactions, time stamp, etc) into one long string, then append a number to it (called a ***nonce***), take the cryptographic hash and see if the hash has the requisite number of “0”s at the front. If not enough “0”s are present, then the nonce is incremented and a new hash is taken. The need for trial and error means that the transaction processor has proven a quantifiable amount of work, earning the right to sign the block and claim the block reward. This process of trying to find a block hash with enough “0”s is called ***mining***.

Importantly, anyone can join the race to find a nonce that produces the requisite number of zeros, making the act of becoming a transaction processor competitive. The work investment of the eventual winner provides the incentive to act honestly, meaning that winners will faithfully validate transaction that they include in the block they sign. Moreover, to maintain the best value for the reward they claim, when they do not win the mining race, miners will have incentive to accept the blocks of those who do win the reace.

This scheme of blockchain security is called ***proof-of-work*** (PoW). The central concept of PoW is that it is not the act of signing a block but the proven work that earns a reward. From the perspective of the block signer, the signature is merely a necessary formality to prove one's identity and claim the reward. From the perspective of the network, the signature serves the important function of certifying the block so that others accept the block into their copies of the blockchain. The block shown in Figure 5 is a PoW block.

Proof-of-Stake

PoW is not the only way to prove eligibility to sign a block. In some systems of blockchain security, the requirement for work is replaced with the requirement that one holds coin. In this type of system, called ***proof-of-stake*** (PoS) those who have greater ownership have an easier time trying nonces because the number of “0”s is inversely proportional to the amount of stake. To prevent the largest holder from signing every block, coin is considered to have an age, which resets when it is sent or used as stake. To

be used as stake, coin must have a minimum age. This means that if the largest holder uses all their stake to easily sign a block, this holder must wait until their stake ages, giving others a chance to sign a block. A PoS system is therefore competitive, but the competition is overwhelmingly dominated by the amount of stake one owns and not the amount of work one is capable of producing.

To prove stake, the ***staker*** (one who stakes coins) effectively sends coins to themselves. Because sending requires a signature, the signature is proof of the stake. The sending transaction also serves to mark the time at which the age of the coin (***coin-age***) resets. In PoS, this staking transaction is called ***coinstake*** and serves the same purpose as a coinbase transaction in that block rewards are claimed therein. For example, if Alice stakes 100 coins to claim a 10 coin block reward, then Alice would create stake inputs worth 100 coins and outputs, called ***mint***, worth 110 coins. The result is that in PoS blocks, the coinbase transaction still comes first but is empty (marking the block as a PoS block) and the staking transaction is second. A PoS block similar to those from the cryptocurrency Peercoin [6] is shown in Figure 9.

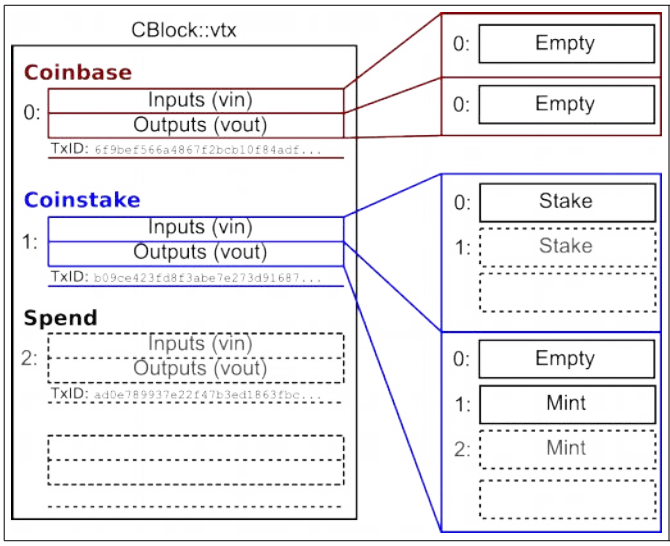

Figure 9. Peercoin-like proof-of-stake blocks have an extra transaction known as coinstake. These proof-of-stake blocks are marked with empty coinbase inputs and outputs and an empty first coinstake output. Spending transactions begin with the third transaction in *vtx*. For all transactions, the input and output sets, called *vin* and *vout*, respectively, are never empty. These sets will at least contain one member. If these members are used as markers, the members may be empty. The total of the mint outputs are are equal to the stake inputs plus the block reward.

In Peercion and many other proof-of-stake cryptocurrencies, stakers are not permitted to claim transaction fees. This does not mean that senders do not need to pay fees. It means that the fees are never claimed and disappear. In this situation, fees are said to “go to the network”, which means that when the fees disappear it reduces the money supply, presumably making the rest of the money supply just a little more valuable. More often than not, this reduction in the coin supply is more than offset by block rewards.

All decentralized ledgers have a challenge wherein it is possible for two different miners or stakers to produce two blocks that link to the same earlier block. This situation represents a ***loss of consensus*** called a ***fork***, shown in Figure 10.

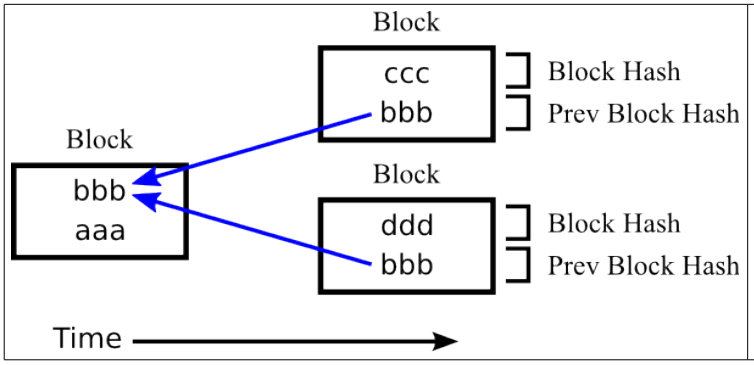

Figure 10. A fork occurs when two otherwise valid blocks are produced that both point to the same previous block. Both blocks `ccc` and `ddd` point back to block `bbb` as their immediate predecessor in a situation called a *fork*. Forks represent a loss of consensus until one side of the fork grows into the longest chain.

Forks happen with regularity (and are just as readily resolved through the consensus mechanism) for all decentralized ledgers. However, for PoS, forks are theoretically very problematic because a staker can attempt to add blocks to both sides of a fork simultaneously, further contributing to the loss of consensus. This possibility for exacerbated loss of consensus is often referred to as ***the nothing-at-stake problem***. Because stake is effectively reproduced on both sides of a fork, it requires no resources to attempt to stake on both sides. *The nothing-at-stake problem* is given by many PoW proponents as a cardinal reason that PoW is superior to PoS. PoW is immune to the *nothing-at-stake problem* because it costs PoW miners proportionally greater resources to mine on multiple sides of a fork simultaneously.

Hybrid Proof-of-Work/Proof-of-Stake

To address the nothing-at-stake problem, many cryptocurrencies, like Peercoin, combine both proof-of-work and proof-of-stake. The combination aims to addresses several potential issues. For example, proof-of-work is theoretically susceptible to a “majority attack” that could occur when a single entity controls a majority of the mining capacity that secures a particular coin. Proof-of-stake is potentially “centralized” because a single entity could eventually purchase ownership of greater than 50% of the money supply. A hybrid system may mitigate these issues. Specifically, a successful majority attack on a hybrid PoW/PoS currency would require both ownership of a majority of the money supply and control of a majority of the mining capacity.

Hybrid PoW/PoS systems do not have specialized block structures. Instead, PoW and PoS blocks are intermingled, typically at about a 1:1 ratio. In other words, a given block in a hybrid PoW/PoS system is either a PoW or a PoS block. Typically, hybrid PoW/PoS block puzzles, though identical in design, are independent for PoW and PoS. This means that workers attempt to solve one problem while stakers attempt to solve a different problem and the two types of blocks are added to the chain more or less independently. Despite this idenpendence, all blocks are added to the same chain. For example, a sequence of blocks from a hybrid PoW/PoS might be something like:

$$\text{PoW} \leftarrow \text{PoS} \leftarrow \text{PoS} \leftarrow \text{PoW} \leftarrow \text{PoW} \leftarrow \text{PoS} \leftarrow \text{PoW} \leftarrow \text{PoS} \leftarrow \ldots$$

with no real limitations on the sequence except an approximate equal probability of the each block's being PoW or PoS.

Tokens and Colored Coins

Several cryptocurrency systems support the creation of ***token currencies*** that exist along with the ***principal currency*** of their block chains. The principal currency is the currency in which transaction fees are paid and without which the token currencies could not function. Token currencies may be thought of as unique currencies that can be created by users. Once created, token currencies are essentially independent of the principal currency, except for reliance on the essential functions of transaction fees and blockchain security.

Token currencies can be created in most of the cryptocurrency systems that use a state ledger, including Ethereum, Nxt, Counterparty and BitShares. Except for Ethereum, which allows for tremendous flexibility with smart contracts, the ability to stipulate spending requirements for tokens is generally somewhat limited. Additionally, sophisticated interaction and interconversion between tokens and the principal currency is in general not possible.

One exception is BitShares, which has SmartCoins. SmartCoins are currencies pegged to an external (non-blockchain) asset through the use of a price feed. SmartCoins are collateralized in the principal currency, fixing the peg on a specific quantity of the external asset. SmartCoins can be converted to the principal currency where the conversion rate is determined by the price feed.

Aside from pegged currencies in BitShares, token currencies are typically used to represent real-world assets like equity ownership of some enterprise. Different Nxt token currencies (usually called “Nxt assets”) have been used extensively to represent shares, where dividends are paid in proportion to one's holdings.

Colored coins are traceable coins that have unique identities. The concept of a colored coin is often confused with the concept of a token currency, mostly because in some cases they are equivalent. However the difference can be understood by examining the various physical representations of domestic currencies. For example, each USD dollar bill has printed on it a unique serial number, making dollar bills analogous to colored coins. However, a dollar that is in a typical bank account is indistinguishable from any other dollar in that account because the representation of any dollar in the account is simply the number 1 added to the account balance.

One way that colored coins could be implemented is by the use of token currencies. In this implementation, a unit of one currency is converted to a token currency with a fixed money supply. For example, imagine that one bitcoin is converted to a token with a money supply of 1000, where each unit of the token currency is worth 0.001 BTC. Individual units of the token currency could be spent in whole units or in fractions thereof.

This system is problematic, at least with regards to using bitcoin, which does not support colored coins natively. The problem is that although bitcoin could be converted to a token by destroying the bitcoin, it is impossible to convert the token back to bitcoin because once destroyed, new bitcoin can only be created by mining. Because the colored coin is not convertible to bitcoin, it can easily lose its value. One way to solve this problem is known as side chains, which will be discussed later. However, a

cryptocurrency that natively allows the interconversion of colored coins bypasses the problem altogether, as interconversion would thereby be fairly trivial.

Smart Contracts

In the present paper, a smart contract will be defined as “computer code that enforces an agreement between two or more parties”. Smart contracts were originally described by Nick Szabo, using the vending machine as a primitive example. To introduce the most essential features of a smart contract, this paper will use the metaphor of a bubblegum machine (Figure 11), the most basic vending machine conceivable. A bubblegum machine accepts money and then allows the user to retrieve bubblegum if the correct money is provided. Most bubblegum machines accept money and permit the user to retrieve bubblegum at the same time (while turning the handle), even though this synchrony is not strictly necessary. One could imagine a different type of machine that divided the process into two consecutive steps.

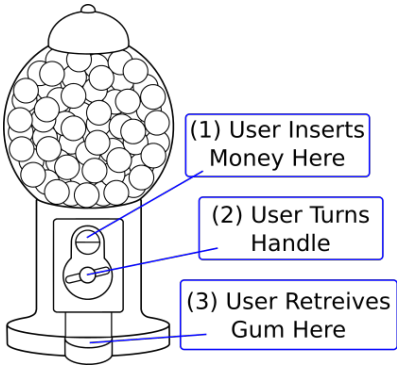

Figure 11. A bubblegum machine is a metaphor for the essential features of a smart contract. The three steps required by the user to obtain bubblegum are to (1) insert money, (2) turn the handle, (3) retrieve the gum. The bubblegum machine has an internal mechanism that decides whether the user has met the conditions of the contract (supplied enough funds in this case). If so, it *permits* the user to retrieve the gum by opening a gate, allowing the gum to fall from the hopper to the candy tray. *Image attributed to http://sweetclipart.com/, modified with labels.*

A bubblegum machine is an excellent metaphor for a smart contract because it has well defined functions that can be expressed as a concise algorithm:

1. Hold a product securely, making the product inaccessible until certain conditions are met.
2. Await a *payment* transaction from a user, in this case the insertion of money.
3. Await a *collection* transaction from the user, in this case turning a handle to retrieve the gum.
4. *Permit* execution of the transaction in (3) if conditions of the sale (contract) are met.

A bubblegum machine has some extraordinary properties. First, it allows two parties to engage in a contract without ever interacting directly. The seller supplied the bubblegum to the hopper and then locked the machine. The buyer inserts money and turns a handle. Second, a bubblegum machine has an internal mechanism that decides, upon action from the buyer, whether conditions are appropriate to allow the transaction to complete. Third, the bubble gum machine is fully passive. It never executes any action in relation to the transaction. It does not withdraw money from the buyer's account nor does it send the money to the seller. It does dispense bubblegum, but that action is initiated (and even fully executed) by the user. It's most critical function is to measure whether the correct the amount of money has been inserted and, if so, *permit* the transaction to complete.

A bubblegum machine might be considered an archetype for what I will term a ***permissive smart contract***, which means its function is to permit transactions rather than execute them. Permissive contracts mirror traditional contracts in that both are passive. For example, the contract that a record company has with a musician does not actually pay the musician. Instead, the record company pays the musician subject to the terms of the contract. Often, as with the recording contract, traditional contracts require parties to take certain actions (the musician to record music and the record company to compensate the musician). Because the blockchain can not compel action of anyone, the bindings to which parties are subject must be reversed for blockchain contracts, often by requiring escrow upfront, then *permitting* the escrow to be withdrawn if the conditions of the contract are satisfied.

Other definitions of smart contracts have been advanced elsewhere, but these definitions include smart contract behaviors that are unnecessary, or even impractical. For example, implicit in many definitions of "smart contract" is the idea that the contract protocol will execute a transaction. Consider a modified version of the ***hedging contract***, or **contract for difference**, as described in the Ethereum white paper.

1. *Wait for party A to input 1000 ether.*
2. *Wait for party B to input 1000 ether.*
3. *Record the USD value of 1000 ether, calculated by querying the data feed contract, in storage, say this is $x.*
4. *After 30 days,* ***the contract expires and the protocol sends*** *$x worth of ether (calculated by querying the data feed contract again to get the new price) to A and the rest to B.*

Although most of this contract is taken verbatim from the Ethereum white paper, the part in bold is new. Here the smart contract has, as part of it's protocol, the act of sending coins to A and B. In practice, though, it is only necessary for the contract to ***permit*** this transaction, not execute it.

At first glance, it may seem as if authority to spend must be vested with the block chain, otherwise one of the parties will simply send the totality of the escrow to himself. This necessity can be eliminated by using a 2-key multisignature (one key belonging to A and one belonging to B), and the contract re-specified:

1. *Wait for party A to send 1000 ether to the multisignature address.*
2. *Wait for party B to input 1000 ether to the multisignature address.*
3. *Record the USD value of 1000 ether, calculated by querying the data feed contract, in storage, say this is $x.*
4. *After 30 days, permit party A to spend $x worth of ether from the multisignature address, and permit party B to spend the rest of the multisignature balance.*

Breakout Chain Proposal for Smart Contracts

Permissive Smart Contracts

It is notable that none of the smart contracts described in the Ethereum white paper necessitate any capacity for the contract to execute a transaction, even though several contracts therein specify this behavior. For example, the crop insurance contract proposed in the Ethereum white paper specifies that, based on the results of a data feed, the farmer may “automatically receive money”. This implies that the smart contract protocol executes the send. However, it is possible to have crop insurance where the farmer, and not the contract, is responsible for executing the transaction. An outline of the contract would be:

1. *The insurer and farmer together register a multisignature address with the contract. Each party has a private key for the public key that they contribute to the multisignature address.*
2. *The insurance agent sends funds covering all but the farmer’s purchase price.*
3. *The farmer buys crop insurance by sending money to the multisignature address.*
4. *After a set duration, the contract permits the farmer to spend (proportional to the size of the policy and inversely proportional to rainfall) from the address registered in Step 1. Likewise the insurer can spend the remaining balance.*

This contract is no more complicated than that found in the Ethereum white paper. Both require (1) the registration of public keys (the payout address for the Ethereum contract and the escrow address, or ***contract address***, for the permissive smart contract), (2) purchase of the insurance, and (3) a transaction that pays the farmer in the case of drought.

Permissive smart contracts do not require any polling of the chain's state. Instead, state is checked only when a transaction is attempted. For example, if the farmer attempts to spend from the registered address after his crops received plenty of rain, then the transaction will be invalidated when network nodes verify the transaction and check the history of the weather feed.

The absence of polling means that the contract does not burn CPU cycles every block. Polling of chain state (e.g. checking whether the contract has matured) undoubtedly still happens, although the farmer is the entity polling and not the miners as they execute the contract.

Specifying Permissive Smart Contracts: Crop Insurance

It is possible to fully specify a crop insurance contract without the execution of any transaction by the protocol itself. The example crop insurance contract in Figure 12 uses a stack-based language called ***contract*** that models Bitcon's *script* language. For this reason, it will be easy for readers familiar with *script* to follow the protocol. Also like Bitcoin's *script* language, the ***contract data stack*** is populated with byte vectors.

Although the native language of permissive smart contracts is the stack-based language *contract*,

higher-level languages .

At the origination of the example crop insurance smart contract, the contract data stack contains (starting at the top) the *Contract Amount*, *Expiration Time*, *Earliest Claim Time*, *Feed Params*, *Max Total Rain*, and *Insured Address*. Deeper members of the contract data stack are not shown, such as the insurer's address. The *Expiration Time* is the time before which the insured may claim a payout. If the *Real Total Rain* exceeds *Max Total Rain*, no claim is possible (tested on line 10). For simplicity, *Feed Params* is meant to represent a group of contract data stack members that are taken as arguments to `GET_FEED_TOTAL`. The `GET_FEED_TOTAL` operator returns the sum of feed observations given a feed identifier, interval and sampling frequency, and others, indicated by the *Feed Params*. A smart contract system would have several different operations that retrieve data from feeds. Examples include `GET_FEED_TOTAL`, `GET_FEED_MEAN`, `GET_FEED_MEDIAN`, and so on. These operations are fine-grained to minimize the complexity of contracts themselves. For this simple contract, the *Contract Amount* is the maximum payout, and is equal to the total provided as escrow by the insurer and as payment by the insured.

When the insurer sets up the contract, the insurer sends escrow funds to the contract address.

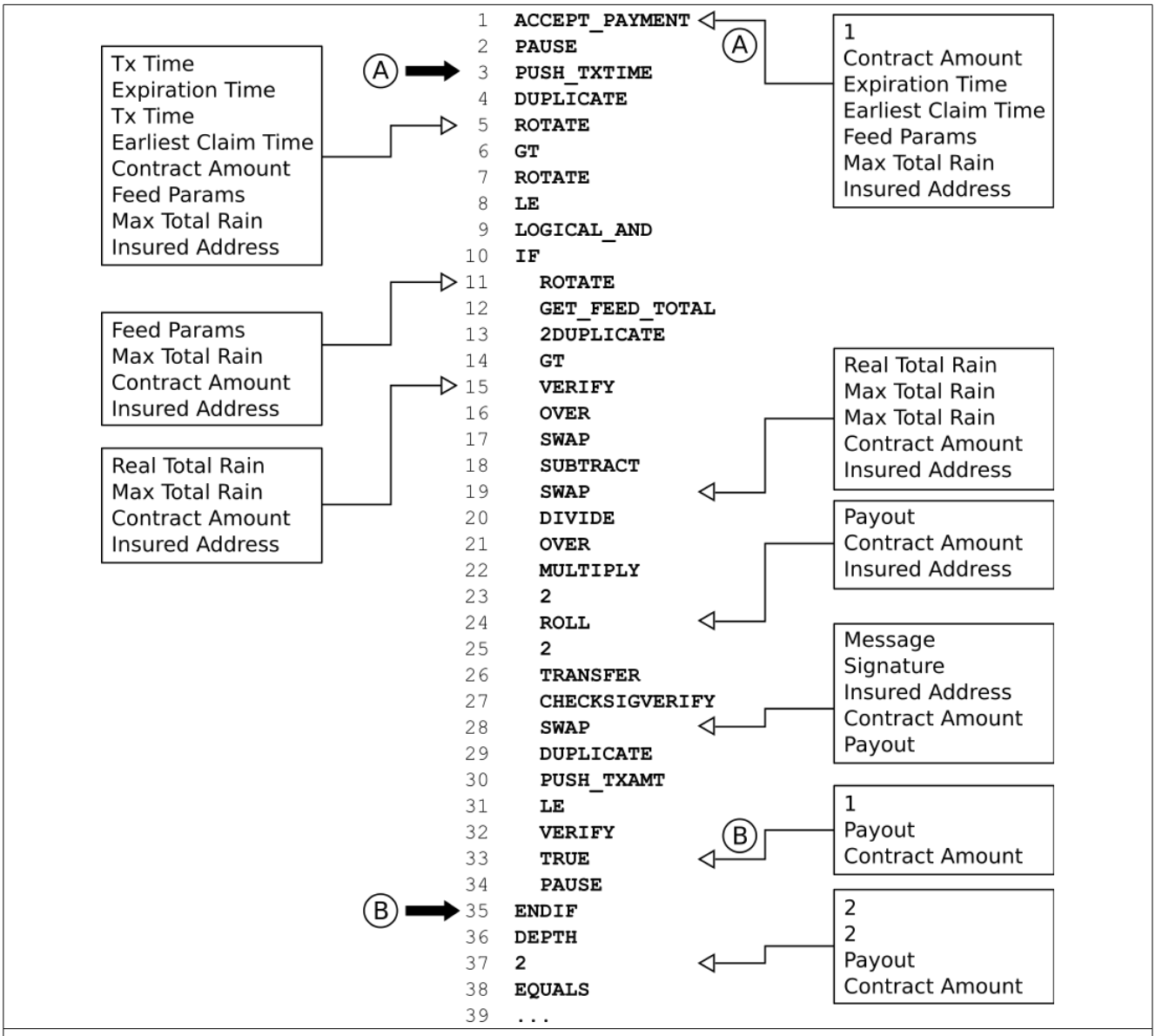

Figure 12. A crop insurance contract can be specified as a permissive contract using a stack-based language. The portion of the contract executed by the insured extends through line 34. Two transactions (circled letters) are represented. Transaction A executes through `PAUSE` at line 2. Transaction B executes through `PAUSE` at line 34. Both transactions terminate with the value 1 at the top of the contract data stack, causing the instruction pointers (filled arrows) to move through `PAUSE`, rather than back up to the previous instruction, as described in the text. The state of the contract data stack upon completion of the corresponding instruction (open arrows) is given in boxes. The stack language, called *contract*, is modeled after Bitcoin's *script* language, with similar instruction names having the same meanings. For example `SWAP` means to switch positions of the top two stack members, `2` means to push the value 2 to the stack. Differences in naming include `LE` (less than or equal to) and other tests for inequality. Other new commands (`ACCEPT_PAYMENT`, `PUSH_TXTIME`, etc.) are explained in the text. This crop insurance contract is purposefully simple for the sake of illustration. For example, it makes no provision if the insured does not pay for the policy, etc.

The example in Figure 12 introduces several new operators. For example, the operator `ACCEPT_PAYMENT` will invalidate a transaction if the transaction attempts to spend from the ***contract address***, which is specified when the contract is created. If the transaction does not try to spend from the contract address, `ACCEPT_PAYMENT` checks to see whether the balance of the contract address is greater than or equal to the top member of the contract data stack (*Contract Amount*). The balance is calculated from all known transactions that execute the smart contract, including the transaction that executes `ACCEPT_PAYMENT`. If the test succeeds, 1 is placed on the top of the contract data stack, 0 otherwise. In all cases, `ACCEPT_PAYMENT` places the input (*Contract Amount*) at the second position of the contract data stack so that the input can be available later if needed.

The `PAUSE` instruction pops the top member of the contract data stack. If this member is 0, the instruction pointer is moved backwards to the instruction immediately prior to `PAUSE`, otherwise the instruction point moves forward to the following instruction. In all cases, execution stops at `PAUSE` and only resumes with a new transaction.

Because *contract* will support the use of subroutines, `PAUSE` is a way to achieve recursion and therefore a way to achieve Turing completeness of permissive smart contracts. One limitation is that each cycle can only be initiated with a new transaction. No single transaction can cause the contract to recurse, meaning that the number of operators in a given transaction's instruction sequence is finite and easily determined. The mechanism to prevent recursion by a single transaction is simple and can be envisioned using pseudocode (Figure 13).

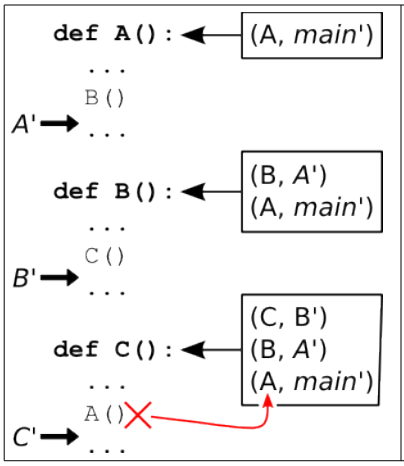

Figure 13. Recursion is prevented within a transaction using the call stack. The pseudocode represents three subroutines `A()`, `B()`, and `C()`. Return addresses after calling each of these functions are *`main'`*, *`B'`*, and *`C'`*, respectively. When each subroutine is called, a 2-tuple of the subroutine address being called (`A`, `B`, `C`) and the return address is pushed to the call stack (boxes). Before calling a subroutine, the runtime searches the call stack for tuples that have the subroutine's address as the first element. If a duplicate address is found, the transaction is invalidated. In the image, the transaction is invalidated at `A()` when called from within `C()` because the bottommost member of the call stack has the address `A` as the tuple's first element.

The Transaction Chain

Each permissive smart contract is executed by a chain of transactions (the ***transaction chain***) that has a strict ordering. This ordering is established by each transaction's having a pointer to the TxID of the previous transaction in the chain. Each permissive smart contract has one and only one transaction chain and no transaction may be a part of more than one chain. The transaction chain is therefore a linked list similar to the blockchain (where each block links to one and only one previous block). The transaction chain does not branch, meaning that when two transactions link to the same previous

transaction in the chain, they cannot both be valid. When a miner receives two transactions that both link to the same previous transaction in the chain, the miner may decide which to include in a block. It is therefore up to clients who execute a contract to keep track of network transactions and link to the newest of the transaction chain.

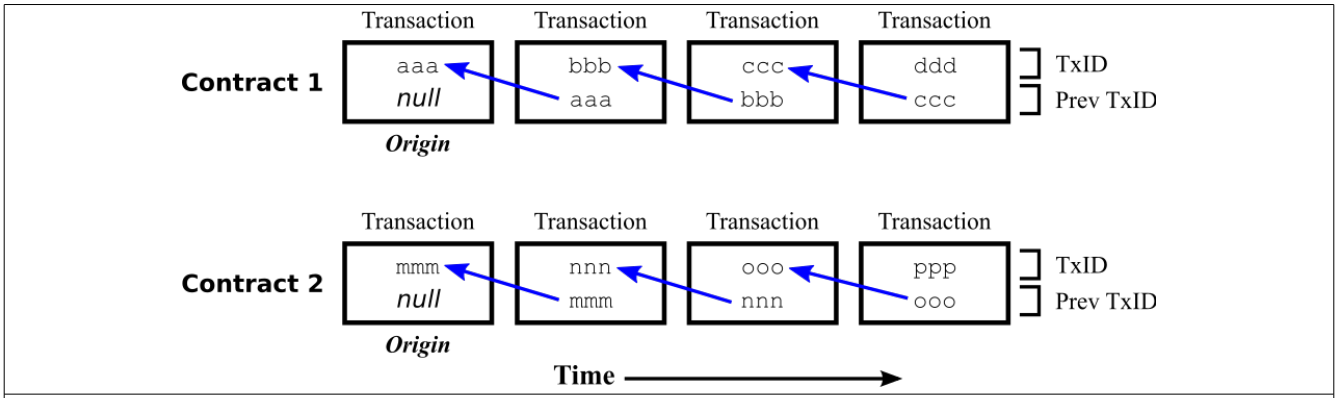

Figure 14. Contracts are executed by transaction chains. The transaction chains for two contracts ("Contract 1" and "Contract 2") are represented. The strings "aaa", "bbb", "ccc", "ddd" represent TxIDs. Each transaction is identified by and contains a record of its own TxID. Each transaction also contains the TxID of the previous transaction in the chain. The original transaction of the chain points to no previous transaction, represented by *null*.

The Data Structures of Permissive Smart Contracts

Permissive contracts can interact with various data structures that belong to the contract itself. For example, the amount and time of the ***current transaction*** can be obtained using `PUSH_TXAMT` and `PUSH_TXTIME`, respectively. The current transaction is the transaction that triggers execution of a segment of the instruction sequence. It follows that each transaction will trigger execution of some segment of the sequence (starting at the beginning or where the last transaction left off) and that execution may or may not complete for a particular transaction. The latter is the case, in the crop insurance example, if a transaction did not transfer enough funds to advance past `ACCEPT_PAYMENT` instruction followed immediately by `PAUSE`.

Permissive smart contracts are created with an ***originating transaction*** which, like other transactions that execute the contract, stores the state of the contract. This storage consists of:

1. The instruction sequence that specifies the contract protocol (Figure 12). For non-originating transactions, this sequence is empty because the contract operations are fully specified by the originating transaction.
2. The instruction pointer that indicates the first instruction to execute for the *next* transaction. For the originating transaction, this pointer points to the first instruction. In Figure 12, the solid arrows are instruction pointers.
3. State of the contract data stack. For the originating transaction, the contract data stack is filled

with starting values. For non-originating transactions, the state of the contract data stack, as it would be after the transaction's execution, is specified. Only the changes necessary to convert the state of contract data stack of the previous transaction to that of the current transaction

4. A pointer to the prior transaction in the contract's transaction chain. For the originating contract, this pointer is null. Given this pointer, it is possible to backtrack to the originating transaction to determine retrieve the instruction sequence and contract address.
5. A ***transaction data stack***. For the originating transaction, the transaction data stack will be empty. For non-originating transactions, the transaction data stack will hold additional data that can be transferred to the contract data stack using `TRANSFER`. As the sole argument, `TRANSFER` takes the number of elements from the transaction data stack to transfer. The `TRANSFER` operator removes the elements from the transaction data stack. The `TRANSFER` operator is used Other than `TRANSFER`, no operators of *contract* can be applied to the transaction data stack. The transaction data stack allows transactions to supply information to the contract such as cryptographic proofs of ownership.
6. State of a call stack. For the originating transaction, the call stack is empty. For non-originating transactions the call, the changes necessary to convert the state of the call stack from the previous transaction to that of the current transaction.

Elements of Flow Control in Permissive Smart Contracts

One question that may arise when looking at the crop insurance contract in Figure 12 is how the contract ensures that actors have the correct authority to execute specific parts of the contract. We can see how this works within the `IF` block. Here, the actor provides a message and signature of that message on the transaction data stack and it is used in `CHECKSIGVERIFY` with the *Insured Address*. If `CHECKSIGVERIFY` fails, the transaction is not valid and the contract state does not change from that specified by the preceding transaction in the chain.

Following the check on lines 36-38 whether the insured has claimed a payout, the contract will check whether the transaction is executed by the insurer, again using `CHECKSIGVERIFY`. This part of the contract is not shown for brevity.

One thing to note is that smart contracts offer two levels of control. The first is that the contract may require proofs different from those required of the transaction script, which could be a pay-to-pubkey-hash, pay-to-script-hash, and so on. The second level of control is in the transaction script, which applies to all transactions, whether they are in a contract transaction chain or not.

It should be noted that it is not necessary to persistently store the state of `IF-ELSE-ENDIF` blocks. The reason is that the blocks subject to `IF` and `ELSE` conditionals that fail are never evaluated. Consider the following block of code:

```
1   FALSE
2   IF
3      {block A}
4   ELSE
5      {block B}
6      PAUSE
7      {block C}
8   ENDIF
```

Here, *block A* at line 3 is skipped, then *block B* is executed. Execution is paused at line 6 and the next transaction, naïve of the `IF-ELSE-ENDIF` nesting, begins executing *block C* at line 7.

To put it simply, whatever instruction at which a transaction begins is *de facto* part of the execution sequence, this includes any instructions immediately following the next `ENDIF`. Similarly, any instruction within an `ELSE` block that is naïvely encountered will not be executed. This later situation can be seen clearly in the following block of code, considering circumstances where execution begins naïvely after the `PAUSE` on line 6.

```
1   TRUE
2   IF
3      {block A}
6      PAUSE
5      {block B}
4   ELSE
7      {block C}
8   ENDIF
```

State in Permissive Smart Contracts

Permissive smart contracts store the contract state information in the transaction itself. For this reason, the contract state is known *before* the transaction is accepted into the blockchain. Also, the miner and the contract executor (the party that makes the transaction) must agree on the state of the contract after execution. This means that the executor can be assured that the contract state will remain consistent with the executor's expectations.

Ultimately, the party who creates transactions is responsible for determining the state of the permissive smart contract and storing this state in the transaction that executes the contract. This is manifestly different from how state is managed in the Ethereum blockchain, where determining state and storing it is the responsibility of the miner. Moreover, permissive smart contracts store contract state at the transaction level whereas Ethereum stores state at the block level.

Sidechains in Permissive Smart Contracts

The chains formed by permissive smart contracts share several fundamental properties with the underlying input/output chain. First, each step in the chain has a precisely defined set of predecessors. Second, each step of the chain is executed in a deterministic number of steps. Third, the state of the

chain is completely specified by members of the chain and is not influenced by the state of any other chain. Fourth, the entity who executes steps in the chain fully specifies the state of the chain after execution. These properties combine to make Bitcoin amenable to pegged side chains. These properties also mean that permissive smart contracts are amenable to pegged side chains, making Breakout Chain's premissive smart contracts arbitrarily scalable.

Multicurrencies

Breakout Chain offers multiple first-class currencies running on a single decentralized UTXO ledger. Each currency is fully supported by by the Breakout Chain input/output mechanism. Multicurrency support is achieved by a simple extension of outputs where the type of currency is given a numerical identifier, termed ***color***, in reference to colored coins.

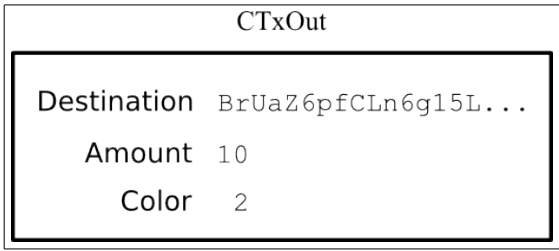

Figure 15. Multicurrency outputs have a currency identifier. The data structure shown is a multicurrency output. The currency identifier (*color*) is 2. Currency identifiers range from 1 to 2,147,483,647, meaning that Breakout Chain can accommodate over 2 billion different currencies.

Having multiple first class currencies on a single block chain allows for novel relationships between currencies. One such relationship incorporated into Breakout Chain is that between Breakout Coin (BRK) and Breakout Stake (BRX). In this relationship, BRX is staked to earn block rewards in BRK. The reason for this relationship is that first coin sale buyers (late 2014) purchased stake and the block reward coin as two separate entities.

To enable this type of relationship, the Breakout Chain PoS block structure (Figure 16) significantly deviates from the Peercoin PoS block structure (Figure 9).

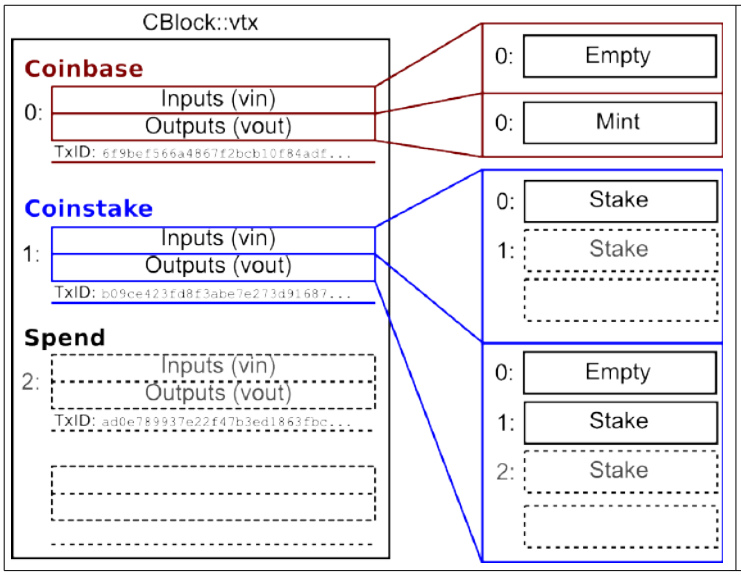

Figure 16. A Breakout Chain PoS block places the mint output in the coinbase transaction. The coinstake inputs and outputs are in the second transaction, where the sum of the input stake must be equal to the sum of the output stake. All new coins are placed in the coinbase transaction. The coinbase transaction is marked with no inputs (empty `vin`), and the coinstake transaction is marked with an empty first output. In Breakout Chain, fees are not collected in PoS blocks.

The Breakout Chain protocol does not allow fees to be collected in PoS blocks although it does require senders to pay a fee. These fees are not lost, however. They can be collected in Breakout Chain PoW blocks (Figure 17) in what is known as ***fee scavenging*** (Figure 18).

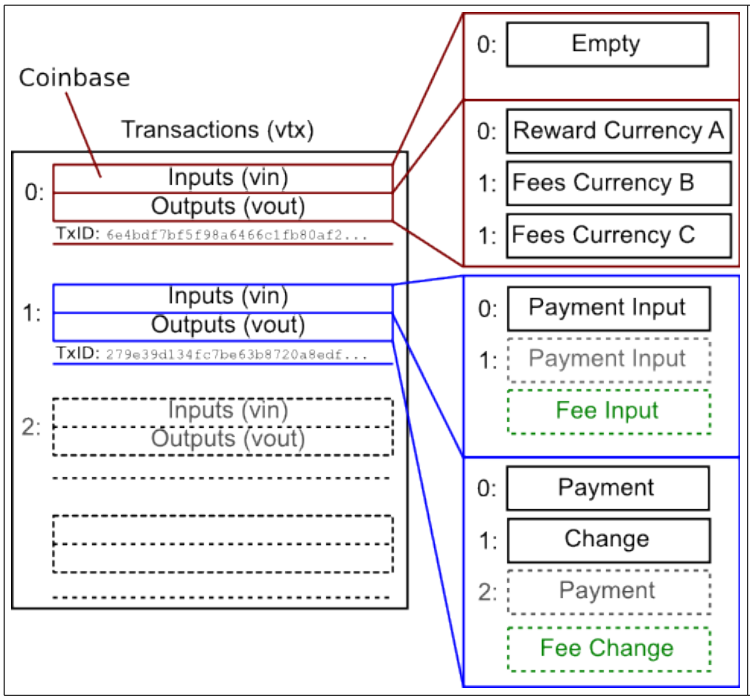

Figure 17. A Breakout Chain PoW block allows fees of different currencies to be collected in the coinbase transaction. Fees in the currency of the block reward are combined with the block reward in the first output of the coinbase transaction. Fees for other currencies follow. Each currency may only appear once in the outputs of the coinbase transaction. When payments are in a different currency than fees (a situation called ***delegated fees***, Figure 19), the fee input (green) may be larger than the required fee, meaning that the sender will write an output for the fee change (green). Transaction inputs and outputs are highly restricted, limited only to a payment currency, and, if different, the fee currency.

If the value of a transaction's outputs for a given currency is less than the value of that transaction's inputs, the difference is available as fees. This difference is subtracted from the money supply. When a PoW miner claims rewards, fee scavenging allows the miner to collect any money not previously claimed in an outputs. The available fee for a given currency is calculated by subtracting the current money supply from the total mint (all coins that have ever been created). The miner for a PoW block is eligible to claim all available fees, restoring the money supply to the total mint. Fee scavenging is illustrated in Figure 18. The purpose of fee scavenging is to add incentive for PoW mining.

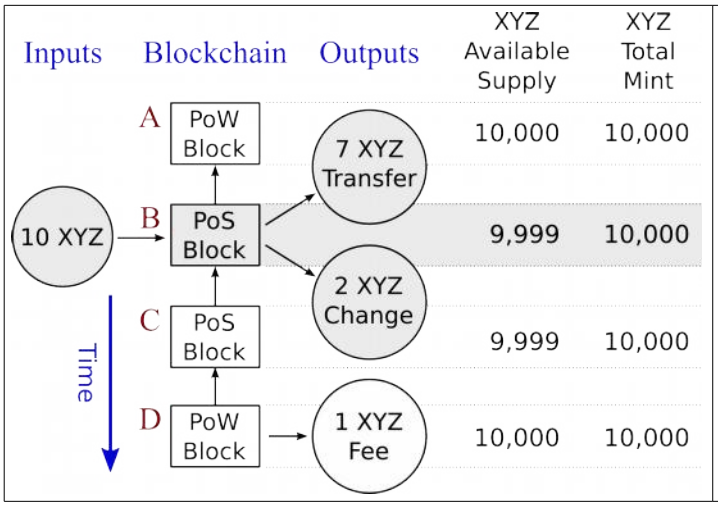

Figure 18. Fee scavenging restores the available money supply of a currency to the total mint. At the end of PoW block A, the total mint for currency XYZ is equal to its available supply of 10,000 XYZ. The transaction in block B (shaded), leaves 1 XYZ unclaimed as required fees, reducing the available supply to 9,999 XYZ. The PoS block C does not give an opportunity to claim the 1 XYZ. PoW block D does, however. The miner of block D claims the 1 XYZ fee in an output, restoring the available supply.

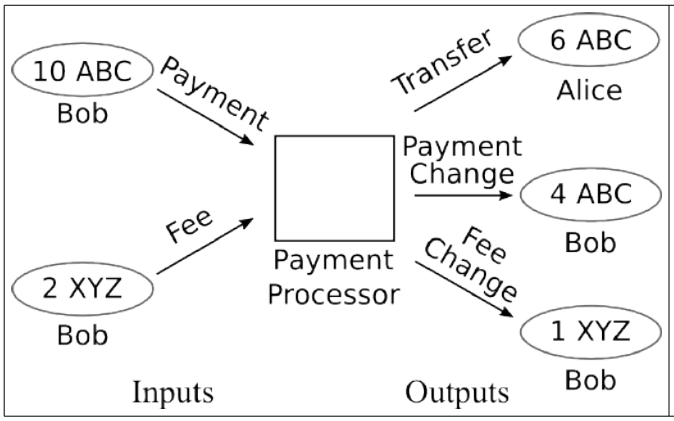

Figure 19. Some currencies have a *fee delegate*, which is a different currency in which fees are paid. Two currencies, ABC and XYZ, are represented. Bob sends 6 ABC to Alice and takes 4 ABC change for himself. To pay the fee, Bob uses currency XYZ, as required. Bob pays 2 XYZ for the fee payment and receives 1 XYZ for the fee change, leaving 1 XYZ to be collected by the miner of the next PoW block, which could also be the block that stores this payment to Alice.

References

[1] https://bitcoin.org/en/bitcoin-paper
[2] https://github.com/ethereum/wiki/wiki/White-Paper
[3] http://szabo.best.vwh.net/smart_contracts_idea.html
[4] https://letstalkbitcoin.com/bitcoin-and-the-three-laws-of-robotics
[5] https://blockstream.com/sidechains.pdf
[6] https://peercoin.net/whitepaper
[7] http://wiki.nxtcrypto.org/wiki/Whitepaper:Nxt
[8] http://counterparty.io/docs/
[9] http://bitsharesblog.com/new-bitshares-whitepaper/