KRABLR:

Secure Decentralised Generalised Peer-to-Peer Scalable Off-Chain Electronic Instant Cash System and MimbleWimble Transaction Ledger

John Biggs

john@techcrunch.com

Abstract

A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

The intent of KRABLR is to create an alternative protocol for building decentralized applications, providing a different set of tradeoffs that we believe will be very useful for a large class of decentralized applications, with particular emphasis on situations where rapid development time, security for small and rarely used applications, and the ability of different applications to very efficiently interact, are important. KRABLR does this by building what is essentially the ultimate abstract foundational layer: a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. A bare-bones version of Namecoin can be written in two lines of code, and other protocols like currencies and reputation systems can be built in under twenty. Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain conditions are met, can also be built on top of the platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state.

The bitcoin protocol can encompass the global financial transaction volume in all electronic payment systems today, without a single custodial third party holding funds or requiring participants to have anything more than a computer using a broadband connection. A decentralized system is proposed whereby transactions are sent

over a network of micropayment channels (a.k.a. payment channels or transaction channels) whose transfer of value occurs off-blockchain. If Bitcoin transactions can be signed with a new sighash type that addresses malleability, these transfers may occur between untrusted parties along the transfer route by contracts which, in the event of uncooperative or hostile participants, are enforceable via broadcast over the bitcoin blockchain in the event of uncooperative or hostile participants, through a series of decrementing timelocks.

Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

Satoshi Nakamoto's development of Bitcoin in 2009 has often been hailed as a radical development in money and currency, being the first example of a digital asset which simultaneously has no backing or "intrinsic value" and no centralized issuer or controller. However, another, arguably more important, part of the Bitcoin experiment is the underlying blockchain technology as a tool of distributed consensus, and attention is rapidly starting to shift to this other aspect of Bitcoin. Commonly cited alternative applications of blockchain technology include using on-blockchain digital assets to represent custom currencies and financial instruments ("colored coins"), the ownership of an underlying physical device ("smart property"), non-fungible assets such as domain names ("Namecoin"), as well as more complex applications involving having digital assets being directly controlled by a piece of code implementing arbitrary rules ("smart contracts") or even blockchain-based "decentralized autonomous organizations" (DAOs). What KRABLR intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code.

Bitcoin is the first widely used financial system for which all the necessary data to validate the system status can be cryptographically verified by anyone. However, it accomplishes this feat by storing all transactions in a public database called "the blockchain" and someone who genuinely wishes to check this state must download the whole thing and basically replay each transaction, check each one as they go. Meanwhile, most of these transactions have not affected the actual final state (they create outputs that are destroyed a transaction later).

At the time of this writing, there were nearly 150 million transactions committed in the blockchain, which must be replayed to produce a set of only 4 million unspent outputs.
It would be better if an auditor needed only to check data on the outputs themselves, but this is impossible because they are valid if and only if the output is at the end of a chain of previous outputs, each signs the next. In other words, the whole blockchain must be validated to confirm the final state.
Add to this that these transactions are cryptographically atomic, it is clear what outputs go into every transaction and what emerges. The "transaction graph" resulting reveals a lot of information and is subjected to analysis by many companies whose business model is to monitor and control the lower classes. This makes it very non-private and even dangerous for people to use.
Some solutions to this have been proposed. Greg Maxwell discovered to encrypt the amounts, so that the graph of the transaction is faceless but still allow validation that the sums are correct. Dr Maxwell also produced CoinJoin, a system for Bitcoin users to combine interactively transactions, confusing the transaction graph. Nicolas van Saberhagen has developed a system to blind the transaction entries, goes much further to cloud the transaction graph (as well as not needed the user interaction). Later, Shen Noether combined the two approaches to obtain "confidential transactions" of Maxwell AND the darkening of van Saberhagen.
These solutions are very good and would make Bitcoin very safe to use. But the problem of too much data is made even worse. Confidential transactions require multi-kilobyte proofs on every output, and van Saberhagen signatures require every output to be stored for ever, since it is not possible to tell when they are truly spent.
Dr. Maxwell's CoinJoin has the problem of needing interactivity. Dr. Yuan Horas Mouton fixed this by making transactions freely mergeable, but he needed to use pairing-based cryptography, which is potentially slower and more difficult to trust. He called this "one-way aggregate signatures" (OWAS).
OWAS had the good idea to combine the transactions in blocks. Imagine that we can combine across blocks (perhaps with some glue data) so that when the outputs are created and destroyed, it is the same as if they never existed. Then, to validate the entire chain, users only need to know when money is entered into the system (new money in each block as in Bitcoin or Monero or peg-ins for sidechains) and final unspent outputs, the rest can be removed and forgotten.
Then we can have Confidential Transactions to hide the amounts and OWAS to blur the transaction graph, and use LESS space than Bitcoin to allow users to fully verify the blockchain. And also imagine that we must not pairing-based cryptography or new hypotheses, just regular discrete logarithms signatures like Bitcoin. Here is what I propose.
I call my creation KRABLR because it is used to prevent the blockchain from talking about all user's information.

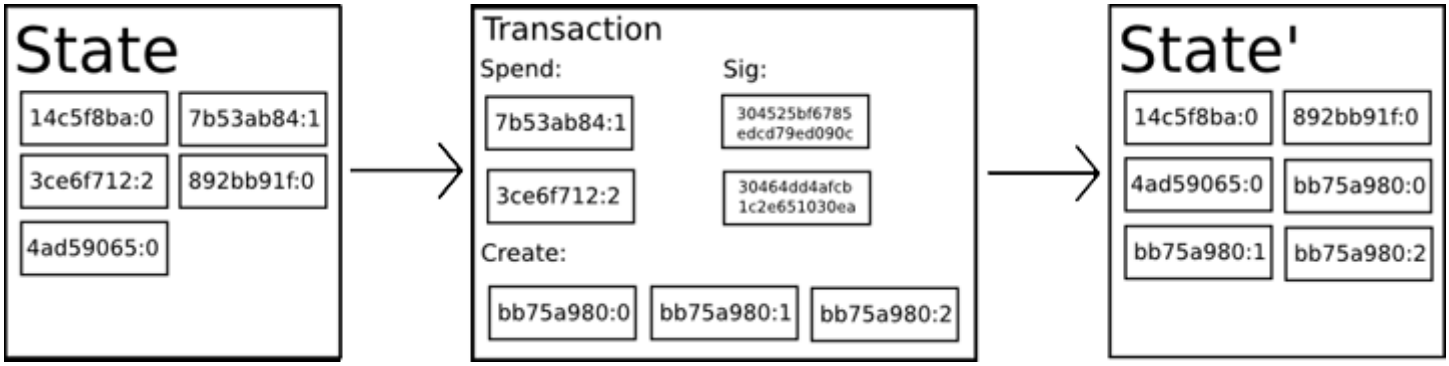
History

The concept of decentralized digital currency, as well as alternative applications like property registries, has been around for decades. The anonymous e-cash protocols of the 1980s and the 1990s were mostly reliant on a cryptographic primitive known as Chaumian Blinding. Chaumian Blinding provided these new currencies with high degrees of privacy, but their underlying protocols largely failed to gain traction because of their reliance on a centralized intermediary. In 1998, Wei Dai's b-money became the first proposal to introduce the idea of creating money through solving computational puzzles as well as decentralized consensus, but the proposal was scant on details as to how decentralized consensus could actually be implemented. In 2005, Hal Finney introduced a concept of "reusable proofs of work", a system which uses ideas from b-money together with Adam Back's computationally difficult Hashcash puzzles to create a concept for a cryptocurrency, but once again fell short of the ideal by relying on trusted computing as a backend. In 2009, a decentralized

currency was for the first time implemented in practice by Satoshi Nakamoto, combining established primitives for managing ownership through public key cryptography with a consensus algorithm for keeping track of who owns coins, known as "proof of work."

The mechanism behind proof of work was a breakthrough because it simultaneously solved two problems. First, it provided a simple and moderately effective consensus algorithm, allowing nodes in the network to collectively agree on a set of updates to the state of the Bitcoin ledger. Second, it provided a mechanism for allowing free entry into the consensus process, solving the political problem of deciding who gets to influence the consensus, while simultaneously preventing Sybil attacks. It does this by substituting a formal barrier to participation, such as the requirement to be registered as a unique entity on a particular list, with an economic barrier - the weight of a single node in the consensus voting process is directly proportional to the computing power that the node brings. Since then, an alternative approach has been proposed called proof of stake, calculating the weight of a node as being proportional to its currency holdings and not its computational resources. The discussion concerning the relative merits of the two approaches is beyond the scope of this paper but it should be noted that both approaches can be used to serve as the backbone of a cryptocurrency.

Bitcoin As A State Transition System



From a technical standpoint, the ledger of a cryptocurrency such as Bitcoin can be thought of as a state transition system, where there is a "state" consisting of the ownership status of all existing bitcoins and a "state transition function" that takes a state and a transaction and outputs a new state which is the result. In a standard banking system, for example, the state is a balance sheet, a transaction is a request to move $X from A to B, and the state transition function reduces the value of A's account by $X and increases the value of B's account by $X. If A's account has less than $X in the first place, the state transition function returns an error. Hence, one can formally define:

```
APPLY(S,TX) -> S' or ERROR
```

In the banking system defined above:

```
APPLY({ Alice: $50, Bob: $50 },"send $20 from Alice to Bob") = { Alice:
$30, Bob: $70 }
```

But:

```
APPLY({ Alice: $50, Bob: $50 },"send $70 from Alice to Bob") = ERROR
```

The "state" in Bitcoin is the collection of all coins (technically, "unspent transaction outputs" or UTXO) that have been minted and not yet spent, with each UTXO having a denomination and an owner (defined by a 20-byte address which is essentially a cryptographic public key). A transaction contains one or more inputs, with each input containing a reference to an existing UTXO and a cryptographic signature produced by the private key associated with the owner's address, and one or more outputs, with each output containing a new UTXO for addition to the state.

The state transition function APPLY(S,TX) -> S' can be defined roughly as follows:

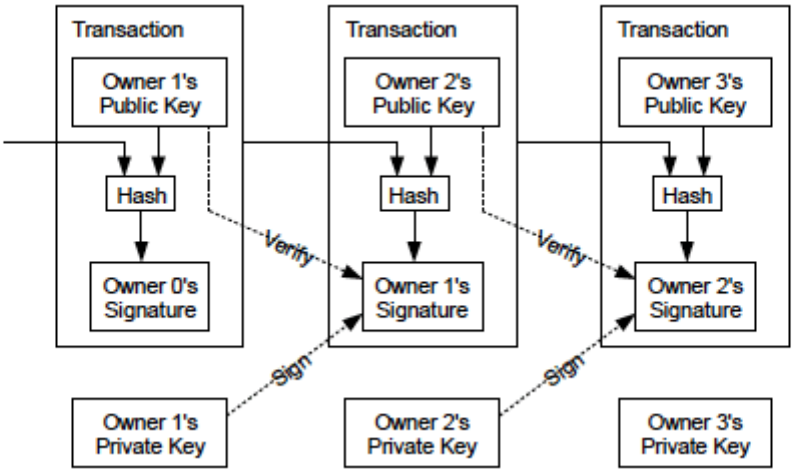
1. For each input in TX: If the referenced UTXO is not in S, return an error. If the provided signature does not match the owner of the UTXO, return an error.
2. If the sum of the denominations of all input UTXO is less than the sum of the denominations of all output UTXO, return an error.
3. Return S' with all input UTXO removed and all output UTXO added.

The first half of the first step prevents transaction senders from spending coins that do not exist, the second half of the first step prevents transaction senders from spending other people's coins, and the second step enforces conservation of value. In order to use this for payment, the protocol is as follows. Suppose Alice wants to send 11.7 BTC to Bob. First, Alice will look for a set of available UTXO that she owns that totals up to at least 11.7 BTC. Realistically, Alice will not be able to get exactly 11.7 BTC; say that the smallest she can get is 6+4+2=12. She then creates a transaction with those three inputs and two outputs. The first output will be 11.7 BTC with Bob's address as its owner, and the second output will be the remaining 0.3 BTC "change". If Alice does not claim this change by sending it to an address owned by herself, the miner will be able to claim it.

Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by

digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.



The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced, and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

Confidential Transactions and OWAS

The first thing we need to do is remove Bitcoin Script. This is sad, but it is too powerful so it is impossible to merge transactions using general scripts. We will demonstrate that confidential transactions of Dr. Maxwell are enough (after some small modification) to authorize spending of outputs and also allows to make combined transactions without interaction. This is in fact identical to OWAS, and allows relaying nodes take some transaction fee or the recipient to change the transaction fees. These additional things Bitcoin can not do, we get for free.

We start by reminding the reader how confidential transactions work. First, the amounts are coded by the following equation:

```
C = r*G + v*H
```

where C is a Pedersen commitment, G and H are fixed nothing-up-my-sleeve elliptic curve group generators, v is the amount, and r is a secret random blinding key.

Attached to this output is a rangeproof which proves that v is in [0, 2^64], so that user cannot exploit the

blinding to produce overflow attacks, etc.
To validate a transaction, the verifier will add commitments for all outputs, plus f*H (f here is the transaction fee which is given explicitly) and subtracts all input commitments. The result must be 0, which proves that no amount was created or destroyed overall.
We note that to create such a transaction, the user must know the sum of all the values of r for commitments entries. Therefore, the r-values (and their sums) act as secret keys. If we can make the r output values known only to the recipient, then we have an authentication system! Unfortunately, if we keep the rule that commits all add to 0, this is impossible, because the sender knows the sum of all _his_ r values, and therefore knows the receipient's r values sum to the negative of that. So instead, we allow the transaction to sum to a nonzero value k*G, and require a signature of an empty string with this as key, to prove its amount component is zero.
We let transactions have as many k*G values as they want, each with a signature, and sum them during verification.

To create transactions sender and recipient do following ritual:

```
1. Sender and recipient agree on amount to be sent. Call this b.

2. Sender creates transaction with all inputs and change output(s), and
gives recipient the total blinding factor (r-value of change minus
r-values of inputs) along with this transaction. So the commitments sum to
r*G - b*H.

3. Recipient chooses random r-values for his outputs, and values that sum
to b minus fee, and adds these to transaction (including range proof).
Now the commitments sum to k*G - fee*H for some k that only recipient
knows.

4. Recipient attaches signature with k to the transaction, and the
explicit fee. It has done.
```

Now, creating transactions in this manner supports OWAS already. To show this, suppose we have two transactions that have a surplus k1*G and k2*G, and the attached signatures with these. Then you can combine the lists of inputs and outputs of the two transactions, with both k1*G and k2*G to the mix, and voilÃ¡! is again a valid transaction. From the combination, it is impossible to say which outputs or inputs are from which original transaction.

Because of this, we change our block format from Bitcoin to this information:

```
1. Explicit amounts for new money (block subsidy or sidechain peg-ins)
with whatever else data this needs. For a sidechain peg-in maybe it
references a Bitcoin transaction that commits to a specific excess k*G
value?

2. Inputs of all transactions
```

```
3. Outputs of all transactions

4. Excess k*G values for all transactions
```

Each of these are grouped together because it do not matter what the transaction boundaries are originally. In addition, Lists 2 3 and 4 should be required to be coded in alphabetical order, since it is quick to check and prevents the block creator of leaking any information about the original transactions.
Note that the outputs are now identified by their hash, and not by their position in a transaction that could easily change. Therefore, it should be banned to have two unspent outputs are equal at the same time, to avoid confusion.

Merging Transactions Across Blocks

Now, we have used Dr. Maxwell's Confidential Transactions to create a noninteractive version of Dr. Maxwell's CoinJoin, but we have not seen the last of marvelous Dr. Maxwell! We need another idea, transaction cut-through, he described in. Again, we create a noninteractive version of this, and to show how it is used with several blocks.
We can imagine now each block as one large transaction. To validate it, we add all the output commitments together, then subtracts all input commitments, k*G values, and all explicit input amounts times H. We find that we could combine transactions from two blocks, as we combined transactions to form a single block, and the result is again a valid transaction. Except now, some output commitments have an input commitment exactly equal to it, where the first block's output was spent in the second block. We could remove both commitments and still have a valid transaction. In fact, there is not even need to check the rangeproof of the deleted output.
The extension of this idea all the way from the genesis block to the latest block, we see that EVERY nonexplicit input is deleted along with its referenced output. What remains are only the unspent outputs, explicit input amounts and every k*G value. And this whole mess can be validated as if it were one transaction: add all unspent commitments output, subtract the values k*G, validate explicit input amounts (if there is anything to validate) then subtract them times H. If the sum is 0, the entire chain is good.

What is this mean? When a user starts up and downloads the chain he needs the following
data from each block:

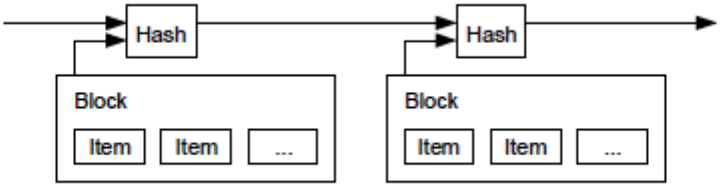
```
1. Explicit amounts for new money (block subsidy or sidechain peg-ins)
with whatever else data this needs.

2. Unspent outputs of all transactions, along with a merkle proof that
each output appeared in the original block.

3. Excess k*G values for all transactions.
```

Bitcoin today there are about 423000 blocks, totaling 80GB or so of data on the hard drive to validate

everything. These data are about 150 million transactions and 5 million unspent nonconfidential outputs. Estimate how much space the number of transactions take on a KRABLR chain. Each unspent output is around 3Kb for rangeproof and Merkle proof. Each transaction also adds about 100 bytes: a k*G value and a signature. The block headers and explicit amounts are negligible. Add this together and get 30Gb -- with a confidential transaction and obscured transaction graph!

Timestamp Server

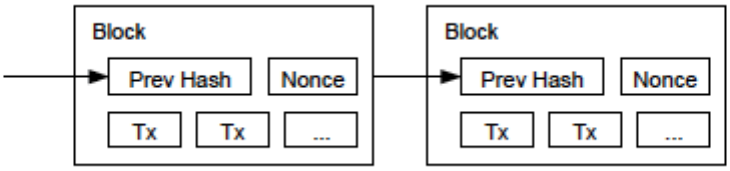
The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.



Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof- of-work system similar to Adam Back's Hashcash, rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.



The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up

diminishes exponentially as subsequent blocks are added.
To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases.

Network

The steps to run the network are as follows:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.
New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

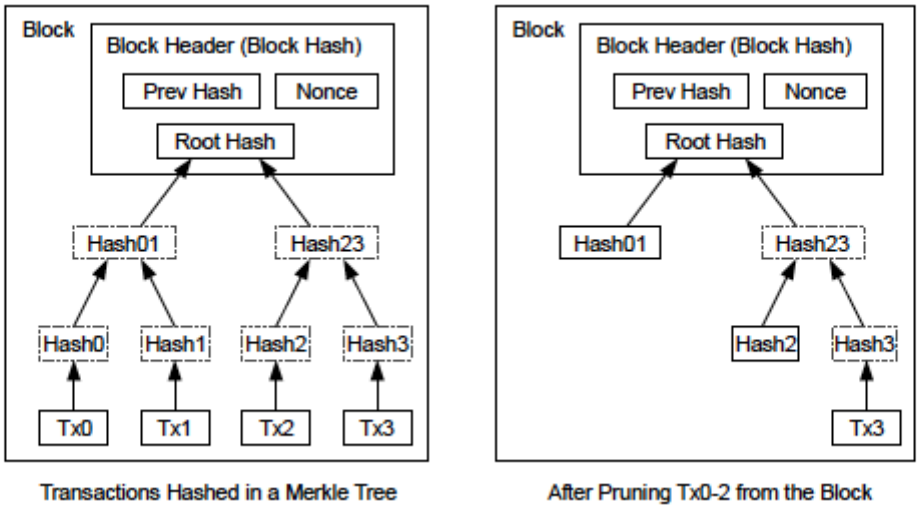
Incentive

By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.
resources to add gold to circulation. In our case, it is CPU time and electricity that is expended. The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction.
Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.
The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be

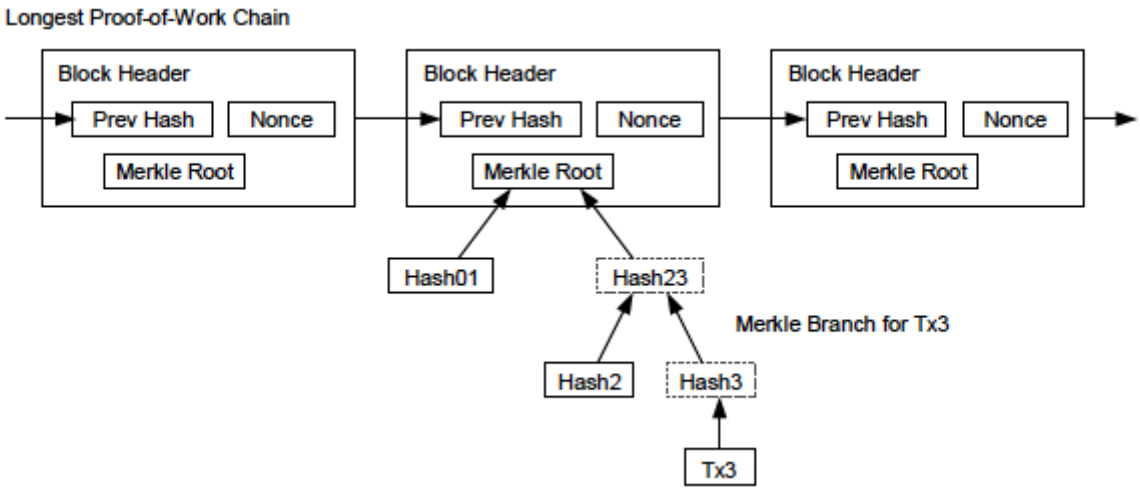
discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree, with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

Simplified Payment Verification

It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.

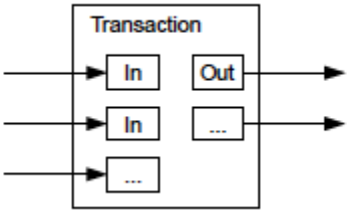


As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can

continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

Combining and Splitting Value

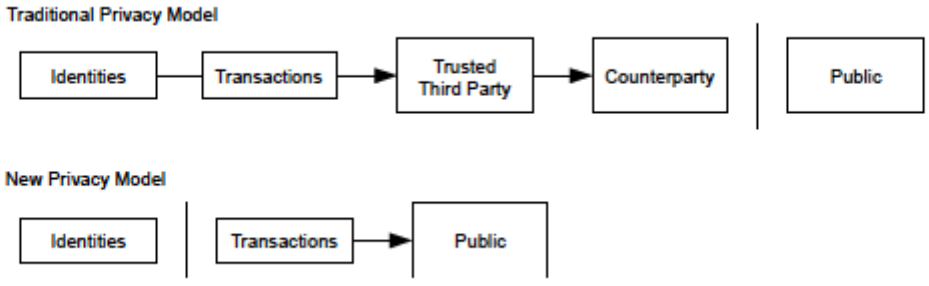
Although it would be possible to handle coins individually, it would be unwieldy to make a separate transaction for every cent in a transfer. To allow value to be split and combined, transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change, if any, back to the sender.



It should be noted that fan-out, where a transaction depends on several transactions, and those transactions depend on many more, is not a problem here. There is never the need to extract a complete standalone copy of a transaction's history.

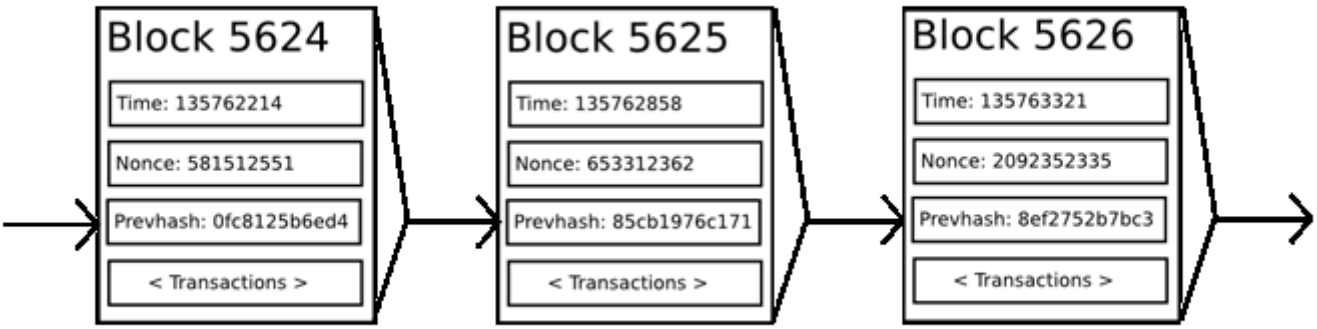
Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.



As an additional firewall, a new key pair should be used for each transaction to keep them from being linked to a common owner. Some linking is still unavoidable with multi-input transactions, which necessarily reveal that their inputs were owned by the same owner. The risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner.

Mining



If we had access to a trustworthy centralized service, this system would be trivial to implement; it could be coded exactly as described, using a centralized server's hard drive to keep track of the state. However, with Bitcoin we are trying to build a decentralized currency system, so we will need to combine the state transition system with a consensus system in order to ensure that everyone agrees on the order of transactions. Bitcoin's decentralized consensus process requires nodes in the network to continuously attempt to produce packages of transactions called "blocks". The network is intended to create one block approximately every ten minutes, with each block containing a timestamp, a nonce, a reference to (i.e., hash of) the previous block and a list of all of the transactions that have taken place since the previous block. Over time, this creates a persistent, ever-growing, "blockchain" that continually updates to represent the latest state of the Bitcoin ledger.

The algorithm for checking if a block is valid, expressed in this paradigm, is as follows:
1. Check if the previous block referenced by the block exists and is valid.
2. Check that the timestamp of the block is greater than that of the previous block and less than 2 hours into the future
3. Check that the proof of work on the block is valid.
4. Let S[0] be the state at the end of the previous block.
5. Suppose TX is the block's transaction list with n transactions. For all i in 0...n-1, set S[i+1] = APPLY(S[i],TX[i]) If any application returns an error, exit and return false.
6. Return true, and register S[n] as the state at the end of this block.

Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block. Additionally, note that the order in which the miner includes transactions into the block matters; if there are two transactions A and B in a block such that B spends a UTXO created by A, then the block will be valid if A comes before B but not otherwise.
The one validity condition present in the above list that is not found in other systems is the requirement for "proof of work". The precise condition is that the double-SHA256 hash of every block, treated as a 256-bit number, must be less than a dynamically adjusted target, which as of the time of this writing is approximately 2^187. The purpose of this is to make block creation computationally "hard", thereby preventing Sybil

attackers from remaking the entire blockchain in their favor. Because SHA256 is designed to be a completely unpredictable pseudorandom function, the only way to create a valid block is simply trial and error, repeatedly incrementing the nonce and seeing if the new hash matches.

At the current target of ~2^187, the network must make an average of ~2^69 tries before a valid block is found; in general, the target is recalibrated by the network every 2016 blocks so that on average a new block is produced by some node in the network every ten minutes. In order to compensate miners for this computational work, the miner of every block is entitled to include a transaction giving themselves 25 BTC out of nowhere. Additionally, if any transaction has a higher total denomination in its inputs than in its outputs, the difference also goes to the miner as a "transaction fee". Incidentally, this is also the only mechanism by which BTC are issued; the genesis state contained no coins at all.

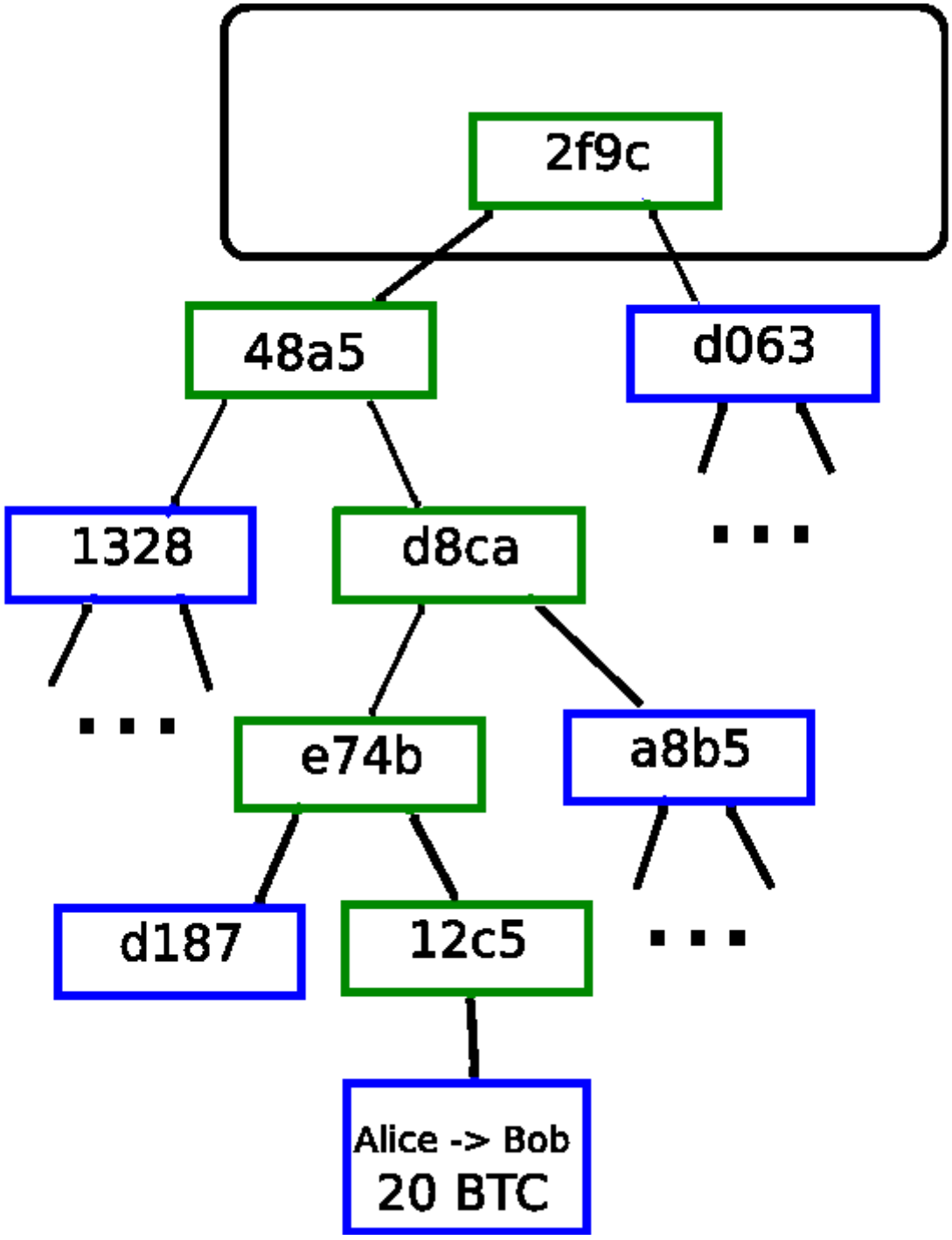
In order to better understand the purpose of mining, let us examine what happens in the event of a malicious attacker. Since Bitcoin's underlying cryptography is known to be secure, the attacker will target the one part of the Bitcoin system that is not protected by cryptography directly: the order of transactions. The attacker's strategy is simple:

1. Send 100 BTC to a merchant in exchange for some product (preferably a rapid-delivery digital good)
2. Wait for the delivery of the product
3. Produce another transaction sending the same 100 BTC to himself

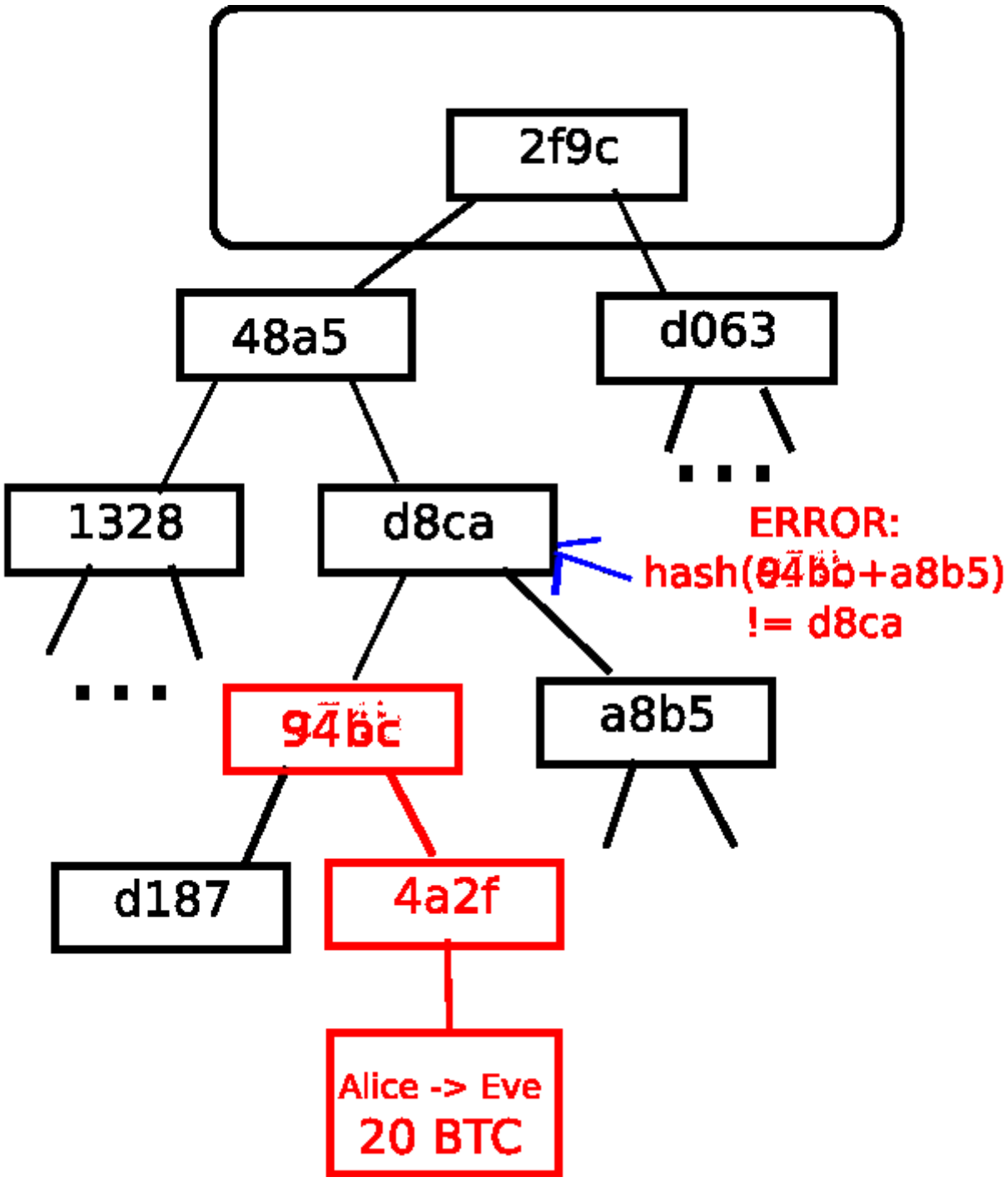
Try to convince the network that his transaction to himself was the one that came first.

Once step (1) has taken place, after a few minutes some miner will include the transaction in a block, say block number 270000. After about one hour, five more blocks will have been added to the chain after that block, with each of those blocks indirectly pointing to the transaction and thus "confirming" it. At this point, the merchant will accept the payment as finalized and deliver the product; since we are assuming this is a digital good, delivery is instant. Now, the attacker creates another transaction sending the 100 BTC to himself. If the attacker simply releases it into the wild, the transaction will not be processed; miners will attempt to run APPLY(S,TX) and notice that TX consumes a UTXO which is no longer in the state. So instead, the attacker creates a "fork" of the blockchain, starting by mining another version of block 270000 pointing to the same block 269999 as a parent but with the new transaction in place of the old one. Because the block data is different, this requires redoing the proof of work. Furthermore, the attacker's new version of block 270000 has a different hash, so the original blocks 270001 to 270005 do not "point" to it; thus, the original chain and the attacker's new chain are completely separate. The rule is that in a fork the longest blockchain is taken to be the truth, and so legitimate miners will work on the 270005 chain while the attacker alone is working on the 270000 chain. In order for the attacker to make his blockchain the longest, he would need to have more computational power than the rest of the network combined in order to catch up (hence, "51% attack").

Merkle Trees




```
01: it suffices to present only a small number of nodes in a Merkle tree
to give a proof of the validity of a branch.
```




```
02: any attempt to change any part of the Merkle tree will eventually lead
to an inconsistency somewhere up the chain.
```

An important scalability feature of Bitcoin is that the block is stored in a multi-level data structure. The "hash" of a block is actually only the hash of the block header, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block. A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to allow the data in a block to be delivered piecemeal: a node can download only the header of a block from one source, the small part of

the tree relevant to them from another source, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward: if a malicious user attempts to swap in a fake transaction into the bottom of a Merkle tree, this change will cause a change in the node above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block (almost certainly with an invalid proof of work).

The Merkle tree protocol is arguably essential to long-term sustainability. A "full node" in the Bitcoin network, one that stores and processes the entirety of every block, takes up about 15 GB of disk space in the Bitcoin network as of April 2014, and is growing by over a gigabyte per month. Currently, this is viable for some desktop computers and not phones, and later on in the future only businesses and hobbyists will be able to participate. A protocol known as "simplified payment verification" (SPV) allows for another class of nodes to exist, called "light nodes", which download the block headers, verify the proof of work on the block headers, and then download only the "branches" associated with transactions that are relevant to them. This allows light nodes to determine with a strong guarantee of security what the status of any Bitcoin transaction, and their current balance, is while downloading only a very small portion of the entire blockchain.

Alternative Blockchain Applications

The idea of taking the underlying blockchain idea and applying it to other concepts also has a long history. In 2005, Nick Szabo came out with the concept of "secure property titles with owner authority", a document describing how "new advances in replicated database technology" will allow for a blockchain-based system for storing a registry of who owns what land, creating an elaborate framework including concepts such as homesteading, adverse possession and Georgian land tax. However, there was unfortunately no effective replicated database system available at the time, and so the protocol was never implemented in practice. After 2009, however, once Bitcoin's decentralized consensus was developed a number of alternative applications rapidly began to emerge.

Namecoin - created in 2010, Namecoin is best described as a decentralized name registration database. In decentralized protocols like Tor, Bitcoin and BitMessage, there needs to be some way of identifying accounts so that other people can interact with them, but in all existing solutions the only kind of identifier available is a pseudorandom hash like 1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy. Ideally, one would like to be able to have an account with a name like "george". However, the problem is that if one person can create an account named "george" then someone else can use the same process to register "george" for themselves as well and impersonate them. The only solution is a first-to-file paradigm, where the first registerer succeeds and the second fails - a problem perfectly suited for the Bitcoin consensus protocol. Namecoin is the oldest, and most successful, implementation of a name registration system using such an idea.

Colored coins - the purpose of colored coins is to serve as a protocol to allow people to create their own digital currencies - or, in the important trivial case of a currency with one unit, digital tokens, on the Bitcoin blockchain. In the colored coins protocol, one "issues" a new currency by publicly assigning a color to a specific Bitcoin UTXO, and the protocol recursively defines the color of other UTXO to be the same as the color of the inputs that the transaction creating them spent (some special rules apply in the case of mixed-color inputs). This allows users to maintain wallets containing only UTXO of a specific color and send them around much like regular bitcoins, backtracking through the blockchain to determine the color of any UTXO that they receive.

Metacoins - the idea behind a metacoin is to have a protocol that lives on top of Bitcoin, using Bitcoin transactions to store metacoin transactions but having a different state transition function, APPLY'. Because the metacoin protocol cannot prevent invalid metacoin transactions from appearing in the Bitcoin blockchain, a rule is added that if APPLY'(S,TX) returns an error, the protocol defaults to APPLY'(S,TX) = S. This provides an easy mechanism for creating an arbitrary cryptocurrency protocol, potentially with advanced features that cannot be implemented inside of Bitcoin itself, but with a very low development cost since the complexities of mining and networking are already handled by the Bitcoin protocol. Metacoins have been used to implement some classes of financial contracts, name registration and decentralized exchange.

Thus, in general, there are two approaches toward building a consensus protocol: building an independent network, and building a protocol on top of Bitcoin. The former approach, while reasonably successful in the case of applications like Namecoin, is difficult to implement; each individual implementation needs to bootstrap an independent blockchain, as well as building and testing all of the necessary state transition and networking code. Additionally, we predict that the set of applications for decentralized consensus technology will follow a power law distribution where the vast majority of applications would be too small to warrant their own blockchain, and we note that there exist large classes of decentralized applications, particularly decentralized autonomous organizations, that need to interact with each other.
The Bitcoin-based approach, on the other hand, has the flaw that it does not inherit the simplified payment verification features of Bitcoin. SPV works for Bitcoin because it can use blockchain depth as a proxy for validity; at some point, once the ancestors of a transaction go far enough back, it is safe to say that they were legitimately part of the state. Blockchain-based meta-protocols, on the other hand, cannot force the blockchain not to include transactions that are not valid within the context of their own protocols. Hence, a fully secure SPV meta-protocol implementation would need to backward scan all the way to the beginning of the Bitcoin blockchain to determine whether or not certain transactions are valid. Currently, all "light" implementations of Bitcoin-based meta-protocols rely on a trusted server to provide the data, arguably a highly suboptimal result especially when one of the primary purposes of a cryptocurrency is to eliminate the need for trust.

Scripting

Even without any extensions, the Bitcoin protocol actually does facilitate a weak version of a concept of "smart contracts". UTXO in Bitcoin can be owned not just by a public key, but also by a more complicated script expressed in a simple stack-based programming language. In this paradigm, a transaction spending that UTXO must provide data that satisfies the script. Indeed, even the basic public key ownership mechanism is implemented via a script: the script takes an elliptic curve signature as input, verifies it against the transaction and the address that owns the UTXO, and returns 1 if the verification is successful and 0 otherwise. Other, more complicated, scripts exist for various additional use cases. For example, one can construct a script that requires signatures from two out of a given three private keys to validate ("multisig"), a setup useful for corporate accounts, secure savings accounts and some merchant escrow situations. Scripts can also be used to pay bounties for solutions to computational problems, and one can even construct a script that says something like "this Bitcoin UTXO is yours if you can provide an SPV proof that you sent a Dogecoin transaction of this denomination to me", essentially allowing decentralized cross-cryptocurrency exchange.

However, the scripting language as implemented in Bitcoin has several important limitations:

Lack of Turing-completeness - that is to say, while there is a large subset of computation that the Bitcoin

scripting language supports, it does not nearly support everything. The main category that is missing is loops. This is done to avoid infinite loops during transaction verification; theoretically it is a surmountable obstacle for script programmers, since any loop can be simulated by simply repeating the underlying code many times with an if statement, but it does lead to scripts that are very space-inefficient. For example, implementing an alternative elliptic curve signature algorithm would likely require 256 repeated multiplication rounds all individually included in the code.

Value-blindness - there is no way for a UTXO script to provide fine-grained control over the amount that can be withdrawn. For example, one powerful use case of an oracle contract would be a hedging contract, where A and B put in $1000 worth of BTC and after 30 days the script sends $1000 worth of BTC to A and the rest to B. This would require an oracle to determine the value of 1 BTC in USD, but even then it is a massive improvement in terms of trust and infrastructure requirement over the fully centralized solutions that are available now. However, because UTXO are all-or-nothing, the only way to achieve this is through the very inefficient hack of having many UTXO of varying denominations (eg. one UTXO of 2^k for every k up to 30) and having O pick which UTXO to send to A and which to B.

Lack of state - UTXO can either be spent or unspent; there is no opportunity for multi-stage contracts or scripts which keep any other internal state beyond that. This makes it hard to make multi-stage options contracts, decentralized exchange offers or two-stage cryptographic commitment protocols (necessary for secure computational bounties). It also means that UTXO can only be used to build simple, one-off contracts and not more complex "stateful" contracts such as decentralized organizations, and makes meta-protocols difficult to implement. Binary state combined with value-blindness also mean that another important application, withdrawal limits, is impossible.

Blockchain-blindness - UTXO are blind to certain blockchain data such as the nonce and previous block hash. This severely limits applications in gambling, and several other categories, by depriving the scripting language of a potentially valuable source of randomness.

Thus, we see three approaches to building advanced applications on top of cryptocurrency: building a new blockchain, using scripting on top of Bitcoin, and building a meta-protocol on top of Bitcoin. Building a new blockchain allows for unlimited freedom in building a feature set, but at the cost of development time, bootstrapping effort and security. Using scripting is easy to implement and standardize, but is very limited in its capabilities, and meta-protocols, while easy, suffer from faults in scalability. With KRABLR, we intend to build an alternative framework that provides even larger gains in ease of development as well as even stronger light client properties, while at the same time allowing applications to share an economic environment and blockchain security.

KRABLR Accounts

In KRABLR, the state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. An KRABLR account contains four fields:

* The nonce, a counter used to make sure each transaction can only be processed once
* The account's current KRB balance

* The account's contract code, if present
* The account's storage (empty by default)

"KRB" is the main internal crypto-fuel of KRABLR, and is used to pay transaction fees. In general, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction; in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.
Note that "contracts" in KRABLR should not be seen as something that should be "fulfilled" or "complied with"; rather, they are more like "autonomous agents" that live inside of the KRABLR execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own KRB balance and their own key/value store to keep track of persistent variables.

Messages and Transactions

The term "transaction" is used in KRABLR to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain:

* The recipient of the message
* A signature identifying the sender
* The amount of KRB to transfer from the sender to the recipient
* An optional data field
* A STARTGAS value, representing the maximum number of computational steps the transaction execution is allowed to take
* A GASPRICE value, representing the fee the sender pays per computational step

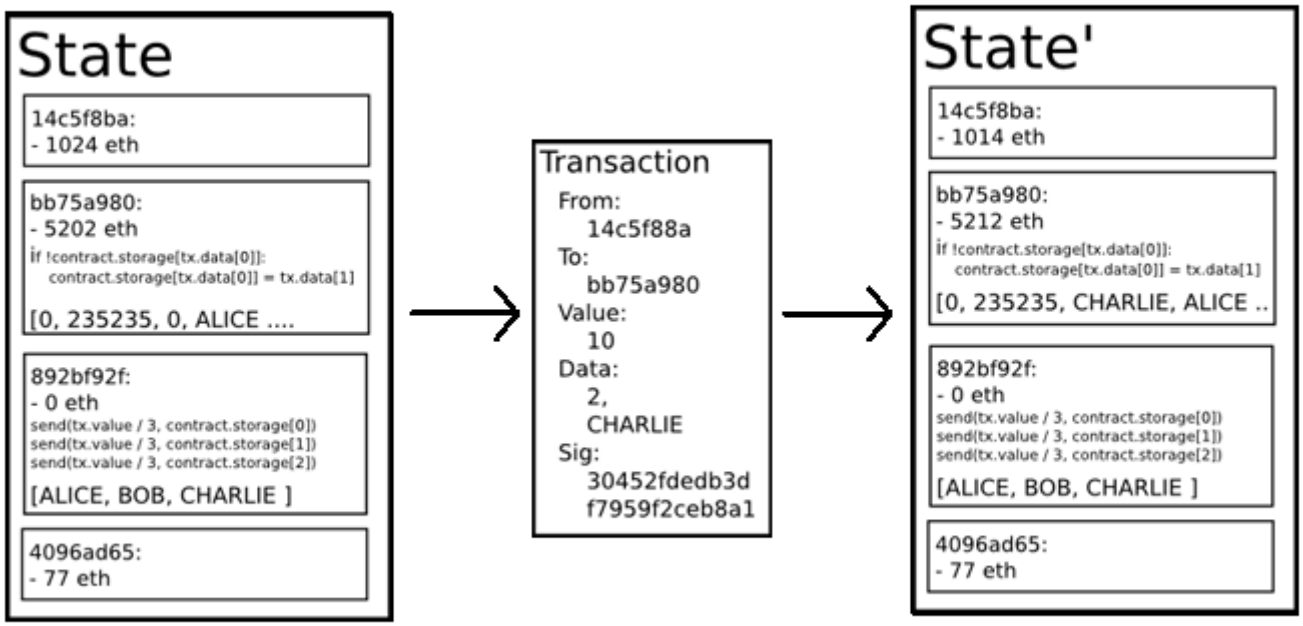
The first three are standard fields expected in any cryptocurrency. The data field has no function by default, but the virtual machine has an opcode with which a contract can access the data; as an example use case, if a contract is functioning as an on-blockchain domain registration service, then it may wish to interpret the data being passed to it as containing two "fields", the first field being a domain to register and the second field being the IP address to register it to. The contract would read these values from the message data and appropriately place them in storage.
The STARTGAS and GASPRICE fields are crucial for KRABLR's anti-denial of service model. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas"; usually, a computational step costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state. There is also a fee of 5 gas for every byte in the transaction data. The intent of the fee system is to require an attacker to pay proportionately for every resource that they consume, including computation, bandwidth and storage; hence, any transaction that leads to the network consuming a greater amount of any of these resources must have a gas fee roughly proportional to the increment.

Contracts have the ability to send "messages" to other contracts. Messages are virtual objects that are never serialized and exist only in the KRABLR execution environment. A message contains:

* The sender of the message (implicit)
* The recipient of the message
* The amount of KRB to transfer alongside the message
* An optional data field
* A STARTGAS value

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the CALL opcode, which produces and executes a message. Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.
Note that the gas allowance assigned by a transaction or contract applies to the total gas consumed by that transaction and all sub-executions. For example, if an external actor A sends a transaction to B with 1000 gas, and B consumes 600 gas before sending a message to C, and the internal execution of C consumes 300 gas before returning, then B can spend another 100 gas before running out of gas.

KRABLR State Transition Function



The KRABLR state transition function, APPLY(S,TX) -> S' can be defined as follows:

1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee as STARTGAS * GASPRICE, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.
3. Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the

transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

For example, suppose that the contract's code is:

```
if !self.storage[calldataload(0)]:
                        self.storage[calldataload(0)] = calldataload(32)
```

Note that in reality the contract code is written in the low-level EVM code; this example is written in Serpent, one of our high-level languages, for clarity, and can be compiled down to EVM code. Suppose that the contract's storage starts off empty, and a transaction is sent with 10 KRB value, 2000 gas, 0.001 KRB gasprice, and 64 bytes of data, with bytes 0-31 representing the number 2 and bytes 32-63 representing the string CHARLIE. The process for the state transition function in this case is as follows:

1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least 2000 * 0.001 = 2 KRB. If it is, then subtract 2 KRB from the sender's account.
3. Initialize gas = 2000; assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more KRB from the sender's account, and add it to the contract's account.
5. Run the code. In this case, this is simple: it checks if the contract's storage at index 2 is used, notices that it is not, and so it sets the storage at index 2 to the value CHARLIE. Suppose this takes 187 gas, so the remaining amount of gas is 1150 - 187 = 963
6. Add 963 * 0.001 = 0.963 KRB back to the sender's account, and return the resulting state.

If there was no contract at the receiving end of the transaction, then the total transaction fee would simply be equal to the provided GASPRICE multiplied by the length of the transaction in bytes, and the data sent alongside the transaction would be irrelevant.
Note that messages work equivalently to transactions in terms of reverts: if a message execution runs out of gas, then that message's execution, and all other executions triggered by that execution, revert, but parent executions do not need to revert. This means that it is "safe" for a contract to call another contract, as if A calls B with G gas then A's execution is guaranteed to lose at most G gas. Finally, note that there is an opcode, CREATE, that creates a contract; its execution mechanics are generally similar to CALL, with the exception that the output of the execution determines the code of a newly created contract.

Code Execution

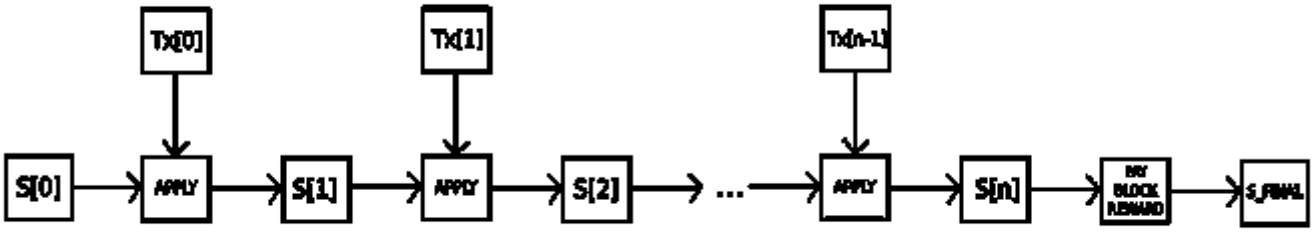
The code in KRABLR contracts is written in a low-level, stack-based bytecode language, referred to as "KRABLR virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte

represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or STOP or RETURN instruction is detected. The operations have access to three types of space in which to store data:

* The stack, a last-in-first-out container to which values can be pushed and popped
* Memory, an infinitely expandable byte array
* The contract's long-term storage, a key/value store. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

The code can also access the value, sender and data of the incoming message, as well as block header data, and the code can also return a byte array of data as an output.
The formal execution model of EVM code is surprisingly simple. While the KRABLR virtual machine is running, its full computational state can be defined by the tuple (block_state, transaction, message, code, memory, stack, pc, gas), where block_state is the global state containing all accounts and includes balances and storage. At the start of every round of execution, the current instruction is found by taking the pcth (Program Counter) byte of code (or 0 if pc >= len(code)), and each instruction has its own definition in terms of how it affects the tuple. For example, ADD pops two items off the stack and pushes their sum, reduces gas by 1 and increments pc by 1, and SSTORE pops the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item. Although there are many ways to optimize KRABLR virtual machine execution via just-in-time compilation, a basic implementation of KRABLR can be done in a few hundred lines of code.

Blockchain and Mining



The KRABLR blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The main difference between KRABLR and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin, KRABLR blocks contain a copy of both the transaction list and the most recent state. Aside from that, two other values, the block number and the difficulty, are also stored in the block. The basic block validation algorithm in KRABLR is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level

KRABLR-specific concepts) are valid.
4. Check that the proof of work on the block is valid.
5. Let S[0] be the state at the end of the previous block.
6. Let TX be the block's transaction list, with n transactions. For all i in 0...n-1, set S[i+1] = APPLY(S[i],TX[i]). If any application returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error.
7. Let S_FINAL be S[n], but adding the block reward paid to the miner.
8. Check if the Merkle tree root of the state S_FINAL is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

The approach may seem highly inefficient at first glance, because it needs to store the entire state with each block, but in reality efficiency should be comparable to that of Bitcoin. The reason is that the state is stored in the tree structure, and after every block only a small part of the tree needs to be changed. Thus, in general, between two adjacent blocks the vast majority of the tree should be the same, and therefore the data can be stored once and referenced twice using pointers (ie. hashes of subtrees). A special kind of tree known as a "Patricia tree" is used to accomplish this, including a modification to the Merkle tree concept that allows for nodes to be inserted and deleted, and not just changed, efficiently. Additionally, because all of the state information is part of the last block, there is no need to store the entire blockchain history - a strategy which, if it could be applied to Bitcoin, can be calculated to provide 5-20x savings in space.
A commonly asked question is "where" contract code is executed, in terms of physical hardware. This has a simple answer: the process of executing contract code is part of the definition of the state transition function, which is part of the block validation algorithm, so if a transaction is added into block B the code execution spawned by that transaction will be executed by all nodes, now and in the future, that download and validate block B.

Token Systems

In general, there are three types of applications on top of KRABLR. The first category is financial applications, providing users with more powerful ways of managing and entering into contracts using their money. This includes sub-currencies, financial derivatives, hedging contracts, savings wallets, wills, and ultimately even some classes of full-scale employment contracts. The second category is semi-financial applications, where money is involved but there is also a heavy non-monetary side to what is being done; a perfect example is self-enforcing bounties for solutions to computational problems. Finally, there are applications such as online voting and decentralized governance that are not financial at all.
On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization. Token systems are surprisingly easy to implement in KRABLR. The key point to understand is that all a currency, or token system, fundamentally is a database with one operation: subtract X units from A and give X units to B, with the proviso that (i) A had at least X units before the transaction and (2) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract.
The basic code for implementing a token system in Serpent looks as follows:

```
def send(to, value):
        if self.storage[msg.sender] >= value:
```

```
                self.storage[msg.sender] = self.storage[msg.sender] - value
                self.storage[to] = self.storage[to] + value
```

This is essentially a literal implementation of the "banking system" state transition function described further above in this document. A few extra lines of code need to be added to provide for the initial step of distributing the currency units in the first place and a few other edge cases, and ideally a function would be added to let other contracts query for the balance of an address. But that's all there is to it. Theoretically, KRABLR-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an KRB balance with which it would refund KRB used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with KRB, but once the KRB is there it would be reusable because the contract would refund it each time.

Financial derivatives

Financial derivatives are the most common application of a "smart contract", and one of the simplest to implement in code. The main challenge in implementing financial contracts is that the majority of them require reference to an external price ticker; for example, a very desirable application is a smart contract that hedges against the volatility of KRB (or another cryptocurrency) with respect to the US dollar, but doing this requires the contract to know what the value of KRB/USD is. The simplest way to do this is through a "data feed" contract maintained by a specific party (eg. NASDAQ) designed so that that party has the ability to update the contract as needed, and providing an interface that allows other contracts to send a message to that contract and get back a response that provides the price.

Given that critical ingredient, the hedging contract would look as follows:

1. Wait for party A to input 1000 KRB.
2. Wait for party B to input 1000 KRB.
3. Record the USD value of 1000 KRB, calculated by querying the data feed contract, in storage, say this is $x.
4. After 30 days, allow A or B to "reactivate" the contract in order to send $x worth of KRB (calculated by querying the data feed contract again to get the new price) to A and the rest to B.

Such a contract would have significant potential in crypto-commerce. One of the main problems cited about cryptocurrency is the fact that it's volatile; although many users and merchants may want the security and convenience of dealing with cryptographic assets, they may not wish to face that prospect of losing 23% of the value of their funds in a single day. Up until now, the most commonly proposed solution has been issuer-backed assets; the idea is that an issuer creates a sub-currency in which they have the right to issue and revoke units, and provide one unit of the currency to anyone who provides them (offline) with one unit of a specified underlying asset (eg. gold, USD). The issuer then promises to provide one unit of the underlying asset to anyone who sends back one unit of the crypto-asset. This mechanism allows any non-cryptographic asset to be "uplifted" into a cryptographic asset, provided that the issuer can be trusted.
In practice, however, issuers are not always trustworthy, and in some cases the banking infrastructure is too

weak, or too hostile, for such services to exist. Financial derivatives provide an alternative. Here, instead of a single issuer providing the funds to back up an asset, a decentralized market of speculators, betting that the price of a cryptographic reference asset (eg. KRB) will go up, plays that role. Unlike issuers, speculators have no option to default on their side of the bargain because the hedging contract holds their funds in escrow. Note that this approach is not fully decentralized, because a trusted source is still needed to provide the price ticker, although arguably even still this is a massive improvement in terms of reducing infrastructure requirements (unlike being an issuer, issuing a price feed requires no licenses and can likely be categorized as free speech) and reducing the potential for fraud.

Identity and Reputation Systems

The earliest alternative cryptocurrency of all, Namecoin, attempted to use a Bitcoin-like blockchain to provide a name registration system, where users can register their names in a public database alongside other data. The major cited use case is for a DNS system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more advanced reputation systems. Here is the basic contract to provide a Namecoin-like name registration system on KRABLR:

```
def register(name, value):
    if !self.storage[name]:
        self.storage[name] = value
```

The contract is very simple; all it is is a database inside the KRABLR network that can be added to, but not modified or removed from. Anyone can register a name with some value, and that registration then sticks forever. A more sophisticated name registration contract will also have a "function clause" allowing other contracts to query it, as well as a mechanism for the "owner" (ie. the first registerer) of a name to change the data or transfer ownership. One can even add reputation and web-of-trust functionality on top.

Decentralized File Storage

Over the past few years, there have emerged a number of popular online file storage startups, the most prominent being Dropbox, seeking to allow users to upload a backup of their hard drive and have the service store the backup and allow the user to access it in exchange for a monthly fee. However, at this point the file storage market is at times relatively inefficient; a cursory look at various existing solutions shows that, particularly at the "uncanny valley" 20-200 GB level at which neither free quotas nor enterprise-level discounts kick in, monthly prices for mainstream file storage costs are such that you are paying for more than the cost of the entire hard drive in a single month. KRABLR contracts can allow for the development of a decentralized file storage ecosystem, where individual users can earn small quantities of money by renting out their own hard drives and unused space can be used to further drive down the costs of file storage.

The key underpinning piece of such a device would be what we have termed the "decentralized Dropbox contract". This contract works as follows. First, one splits the desired data up into blocks, encrypting each block for privacy, and builds a Merkle tree out of it. One then makes a contract with the rule that, every N blocks, the contract would pick a random index in the Merkle tree (using the previous block hash, accessible from contract code, as a source of randomness), and give X KRB to the first entity to supply a transaction with a simplified payment verification-like proof of ownership of the block at that particular index in the tree. When a user wants to re-download their file, they can use a micropayment channel protocol (eg. pay 1 szabo

per 32 kilobytes) to recover the file; the most fee-efficient approach is for the payer not to publish the transaction until the end, instead replacing the transaction with a slightly more lucrative one with the same nonce after every 32 kilobytes.

An important feature of the protocol is that, although it may seem like one is trusting many random nodes not to decide to forget the file, one can reduce that risk down to near-zero by splitting the file into many pieces via secret sharing, and watching the contracts to see each piece is still in some node's possession. If a contract is still paying out money, that provides a cryptographic proof that someone out there is still storing the file.

Decentralized Autonomous Organizations

The general concept of a "decentralized autonomous organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shares; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.

A general outline for how to code a DAO is as follows. The simplest design is simply a piece of self-modifying code that changes if two thirds of members agree on a change. Although code is theoretically immutable, one can easily get around this and have de-facto mutability by having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage. In a simple implementation of such a DAO contract, there would be three transaction types, distinquished by the data provided in the transaction:

* [0,i,K,V] to register a proposal with index i to change the address at storage index K to value V
* [0,i] to register a vote in favor of proposal i
* [2,i] to finalize proposal i if enough votes have been made

The contract would then have clauses for each of these. It would maintain a record of all open storage changes, along with a list of who voted for them. It would also have a list of all members. When any storage change gets to two thirds of members voting for it, a finalizing transaction could execute the change. A more sophisticated skeleton would also have built-in voting ability for features like sending a transaction, adding members and removing members, and may even provide for Liquid Democracy-style vote delegation (ie. anyone can assign someone to vote for them, and assignment is transitive so if A assigns B and B assigns C then C determines A's vote). This design would allow the DAO to grow organically as a decentralized community, allowing people to eventually delegate the task of filtering out who is a member to specialists, although unlike in the "current system" specialists can easily pop in and out of existence over time as individual community members change their alignments.

An alternative model is for a decentralized corporation, where any account can have zero or more shares, and two thirds of the shares are required to make a decision. A complete skeleton would involve asset management functionality, the ability to make an offer to buy or sell shares, and the ability to accept offers (preferably with

an order-matching mechanism inside the contract). Delegation would also exist Liquid Democracy-style, generalizing the concept of a "board of directors".

Further Applications

1. Savings wallets. Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts KRB into a contract with Bob, a bank, as follows:

* Alice alone can withdraw a maximum of 1% of the funds per day.
* Bob alone can withdraw a maximum of 1% of the funds per day, but Alice has the ability to make a transaction with her key shutting off this ability.
* Alice and Bob together can withdraw anything.

2. Crop insurance. One can easily make a financial derivatives contract but using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well. This can be expanded to natural disaster insurance generally.

3. A decentralized data feed. For financial contracts for difference, it may actually be possible to decentralize the data feed via a protocol called "SchellingCoin". SchellingCoin basically works as follows: N parties all put into the system the value of a given datum (eg. the KRB/USD price), the values are sorted, and everyone between the 25th and 75th percentile gets one token as a reward. Everyone has the incentive to provide the answer that everyone else will provide, and the only value that a large number of players can realistically agree on is the obvious default: the truth. This creates a decentralized protocol that can theoretically provide any number of values, including the KRB/USD price, the temperature in Berlin or even the result of a particular hard computation.

4. Smart multisignature escrow. Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. KRABLR allows for more granularity; for example, four out of five can spend everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, KRABLR multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.

5. Cloud computing. The EVM technology can also be used to create a verifiable computing environment, allowing users to ask others to carry out computations and then optionally ask for proofs that computations at certain randomly selected checkpoints were done correctly. This allows for the creation of a cloud computing market where any user can participate with their desktop, laptop or specialized server, and spot-checking together with security deposits can be used to ensure that the system is trustworthy (ie. nodes cannot profitably cheat). Although such a system may not be suitable for all tasks; tasks that require a high level of inter-process communication, for example, cannot easily be done on a large cloud of nodes. Other tasks, however, are much easier to parallelize; projects like SETI@home, folding@home and genetic algorithms can easily be implemented on top of such a platform.

6. Peer-to-peer gambling. Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard

Clayton's Cyberdice, can be implemented on the KRABLR blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash, and more advanced protocols can be built up from there, creating gambling services with near-zero fees that have no ability to cheat.

7. Prediction markets. Provided an oracle or SchellingCoin, prediction markets are also easy to implement, and prediction markets together with SchellingCoin may prove to be the first mainstream application of futarchy as a governance protocol for decentralized organizations.

8. On-chain decentralized marketplaces, using the identity and reputation system as a base.

Modified GHOST Implementation

The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013. The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

As described by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor (in KRABLR jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it. To solve the second issue of centralization bias, we go beyond the protocol described by Sompolinsky and Zohar, and also provide block rewards to stales: a stale block receives 87.5% of its base reward, and the nephew that includes the stale block receives the remaining 12.5%. Transaction fees, however, are not awarded to uncles.

KRABLR implements a simplified version of GHOST which only goes down seven levels. Specifically, it is defined as follows:

* A block must specify a parent, and it must specify 0 or more uncles
* An uncle included in block B must have the following properties:
* -> It must be a direct child of the kth generation ancestor of B, where 2 <= k <= 7.
* -> It cannot be an ancestor of B
* -> An uncle must be a valid block header, but does not need to be a previously verified or even valid block
* -> An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)
* For every uncle U in block B, the miner of B gets an additional 3.125% added to its coinbase reward and the miner of U gets 93.75% of a standard coinbase reward.

This limited version of GHOST, with uncles includable only up to 7 generations, was used for two reasons.

First, unlimited GHOST would include too many complications into the calculation of which uncles for a given block are valid. Second, unlimited GHOST with compensation as used in KRABLR removes the incentive for a miner to mine on the main chain and not the chain of a public attacker.

Fees

Because every transaction published into the blockchain imposes on the network the cost of needing to download and verify it, there is a need for some regulatory mechanism, typically involving transaction fees, to prevent abuse. The default approach, used in Bitcoin, is to have purely voluntary fees, relying on miners to act as the gatekeepers and set dynamic minimums. This approach has been received very favorably in the Bitcoin community particularly because it is "market-based", allowing supply and demand between miners and transaction senders determine the price. The problem with this line of reasoning is, however, that transaction processing is not a market; although it is intuitively attractive to construe transaction processing as a service that the miner is offering to the sender, in reality every transaction that a miner includes will need to be processed by every node in the network, so the vast majority of the cost of transaction processing is borne by third parties and not the miner that is making the decision of whether or not to include it. Hence, tragedy-of-the-commons problems are very likely to occur.
However, as it turns out this flaw in the market-based mechanism, when given a particular inaccurate simplifying assumption, magically cancels itself out. The argument is as follows. Suppose that:

1. A transaction leads to k operations, offering the reward kR to any miner that includes it where R is set by the sender and k and R are (roughly) visible to the miner beforehand.
2. An operation has a processing cost of C to any node (ie. all nodes have equal efficiency)
3. There are N mining nodes, each with exactly equal processing power (ie. 1/N of total)
4. No non-mining full nodes exist.

A miner would be willing to process a transaction if the expected reward is greater than the cost. Thus, the expected reward is kR/N since the miner has a 1/N chance of processing the next block, and the processing cost for the miner is simply kC. Hence, miners will include transactions where $kR/N > kC$, or $R > NC$. Note that R is the per-operation fee provided by the sender, and is thus a lower bound on the benefit that the sender derives from the transaction, and NC is the cost to the entire network together of processing an operation. Hence, miners have the incentive to include only those transactions for which the total utilitarian benefit exceeds the cost.
However, there are several important deviations from those assumptions in reality:

1. The miner does pay a higher cost to process the transaction than the other verifying nodes, since the extra verification time delays block propagation and thus increases the chance the block will become a stale.
2. There do exist nonmining full nodes.
3. The mining power distribution may end up radically inegalitarian in practice.
4. Speculators, political enemies and crazies whose utility function includes causing harm to the network do exist, and they can cleverly set up contracts where their cost is much lower than the cost paid by other verifying nodes.

(1) provides a tendency for the miner to include fewer transactions, and (2) increases NC; hence, these two effects at least partially cancel each other out. (3) and (4) are the major issue; to solve them we simply institute

a floating cap: no block can have more operations than BLK_LIMIT_FACTOR times the long-term exponential moving average. Specifically:

```
blk.oplimit = floor((blk.parent.oplimit * (EMAFACTOR - 1) +
floor(parent.opcount * BLK_LIMIT_FACTOR)) / EMA_FACTOR)
```

BLK_LIMIT_FACTOR and EMA_FACTOR are constants that will be set to 65536 and 1.5 for the time being, but will likely be changed after further analysis.
There is another factor disincentivizing large block sizes in Bitcoin: blocks that are large will take longer to propagate, and thus have a higher probability of becoming stales. In KRABLR, highly gas-consuming blocks can also take longer to propagate both because they are physically larger and because they take longer to process the transaction state transitions to validate. This delay disincentive is a significant consideration in Bitcoin, but less so in KRABLR because of the GHOST protocol; hence, relying on regulated block limits provides a more stable baseline.
KRABLR Network fees, which differ from blockchain fees, are paid directly between participants within the channel. The fees pay for the time-value of money for consuming the channel for a determined maximum period of time, and for counterparty risk of non-communication.
Counterparty risk for fees only exist with one's direct channel counterparty. If a node two hops away decides to disconnect and their transaction gets broadcast on the blockchain, one's direct counterparties should not broadcast on the blockchain, but continue to update via novation with a new Commitment Transaction. See the Decrementing Timelocks entry in the HTLC section for more information about counterparty risk.
The time-value of fees pays for consuming time (e.g. 3 days) and is conceptually equivalent to a gold lease rate without custodial risk; it is the time-value for using up the access to money for a very short duration. Since certain paths may become very profitable in one direction, it is possible for fees to be negative to encourage the channel to be available for those profitable paths.

Computation And Turing-Completeness

An important note is that the KRABLR virtual machine is Turing-complete; this means that EVM code can encode any computation that can be conceivably carried out, including infinite loops. EVM code allows looping in two ways. First, there is a JUMP instruction that allows the program to jump back to a previous spot in the code, and a JUMPI instruction to do conditional jumping, allowing for statements like while x < 27: x = x * 2. Second, contracts can call other contracts, potentially allowing for looping through recursion. This naturally leads to a problem: can malicious users essentially shut miners and full nodes down by forcing them to enter into an infinite loop? The issue arises because of a problem in computer science known as the halting problem: there is no way to tell, in the general case, whether or not a given program will ever halt.
As described in the state transition section, our solution works by requiring a transaction to set a maximum number of computational steps that it is allowed to take, and if execution takes longer computation is reverted but fees are still paid. Messages work in the same way. To show the motivation behind our solution, consider the following examples:

* An attacker creates a contract which runs an infinite loop, and then sends a transaction activating that loop to the miner. The miner will process the transaction, running the infinite loop, and wait for it to run out of gas. Even though the execution runs out of gas and stops halfway through, the transaction is still valid and the miner still claims the fee from the attacker for each computational step.

* An attacker creates a very long infinite loop with the intent of forcing the miner to keep computing for such a long time that by the time computation finishes a few more blocks will have come out and it will not be possible for the miner to include the transaction to claim the fee. However, the attacker will be required to submit a value for STARTGAS limiting the number of computational steps that execution can take, so the miner will know ahead of time that the computation will take an excessively large number of steps.

* An attacker sees a contract with code of some form like send(A,contract.storage[A]); contract.storage[A] = 0, and sends a transaction with just enough gas to run the first step but not the second (ie. making a withdrawal but not letting the balance go down). The contract author does not need to worry about protecting against such attacks, because if execution stops halfway through the changes get reverted.

* A financial contract works by taking the median of nine proprietary data feeds in order to minimize risk. An attacker takes over one of the data feeds, which is designed to be modifiable via the variable-address-call mechanism described in the section on DAOs, and converts it to run an infinite loop, thereby attempting to force any attempts to claim funds from the financial contract to run out of gas. However, the financial contract can set a gas limit on the message to prevent this problem.

The alternative to Turing-completeness is Turing-incompleteness, where JUMP and JUMPI do not exist and only one copy of each contract is allowed to exist in the call stack at any given time. With this system, the fee system described and the uncertainties around the effectiveness of our solution might not be necessary, as the cost of executing a contract would be bounded above by its size. Additionally, Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop, and even that loop could be removed by making 26 repetitions of a one-line piece of code. Given the serious implications of Turing-completeness, and the limited benefit, why not simply have a Turing-incomplete language? In reality, however, Turing-incompleteness is far from a neat solution to the problem. To see why, consider the following contracts:

```
C0: call(C1); call(C1);
C1: call(C2); call(C2);
C2: call(C3); call(C3);
...
C49: call(C50); call(C50);
C50: (run one step of a program and record the change in storage)
```

Now, send a transaction to A. Thus, in 51 transactions, we have a contract that takes up 250 computational steps. Miners could try to detect such logic bombs ahead of time by maintaining a value alongside each contract specifying the maximum number of computational steps that it can take, and calculating this for contracts calling other contracts recursively, but that would require miners to forbid contracts that create other contracts (since the creation and execution of all 26 contracts above could easily be rolled into a single contract). Another problematic point is that the address field of a message is a variable, so in general it may not even be possible to tell which other contracts a given contract will call ahead of time. Hence, all in all, we have a surprising conclusion: Turing-completeness is surprisingly easy to manage, and the lack of Turing-completeness is equally surprisingly difficult to manage unless the exact same controls are in place -

but in that case why not just let the protocol be Turing-complete?

Currency And Issuance

The KRABLR network includes its own built-in currency, KRB, which serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate in Bitcoin), the denominations will be pre-labeled:

* 1: wei
* 10^12: szabo
* 10^15: finney
* 10^18: KRB

This should be taken as an expanded version of the concept of "dollars" and "cents" or "BTC" and "satoshi". In the near future, we expect "KRB" to be used for ordinary transactions, "finney" for microtransactions and "szabo" and "wei" for technical discussions around fees and protocol implementation; the remaining denominations may become useful later and should not be included in clients at this point.
The issuance model will be as follows:

* KRB will be released in a currency sale at the price of 1000-2000 KRB per BTC, a mechanism intended to fund the KRABLR organization and pay for development that has been used with success by other platforms such as Mastercoin and NXT. Earlier buyers will benefit from larger discounts. The BTC received from the sale will be used entirely to pay salaries and bounties to developers and invested into various for-profit and non-profit projects in the KRABLR and cryptocurrency ecosystem.

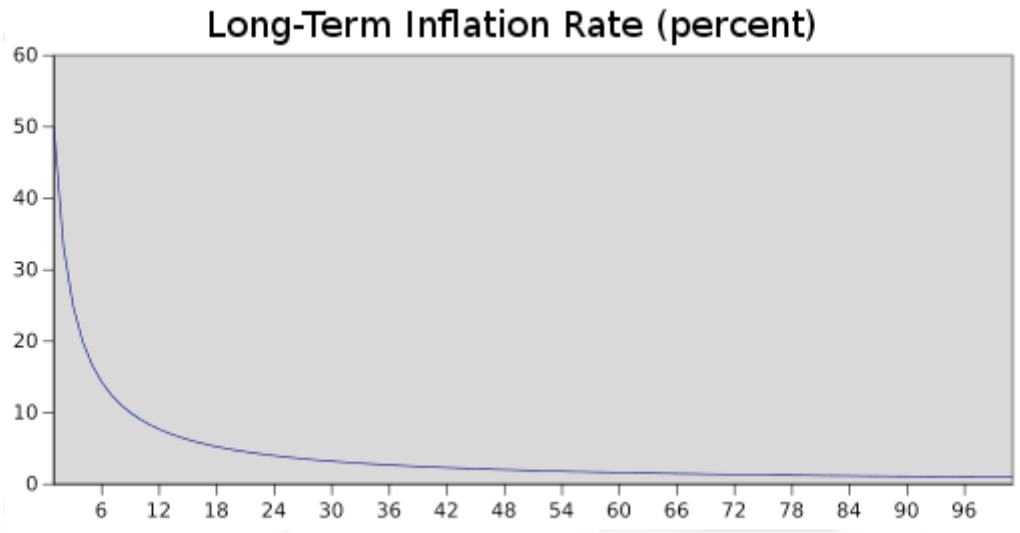
* 0.099x the total amount sold (60102216 KRB) will be allocated to the organization to compensate early contributors and pay KRB-denominated expenses before the genesis block.

* 0.099x the total amount sold will be maintained as a long-term reserve.

* 0.26x the total amount sold will be allocated to miners per year forever after that point.

Group	At launch	After 1 year	After 5 years
Currency units	1.198X	1.458X	2.498X
Purchasers	83.5%	68.6%	40.0%
Reserve spent pre-sale	8.26%	6.79%	3.96%
Reserve used post-sale	8.26%	6.79%	3.96%
Miners	0%	17.8%	52.0%

Long-Term Supply Growth Rate (percent)



`Despite the linear currency issuance, just like with Bitcoin over time the supply growth rate nevertheless tends to zero`

The two main choices in the above model are (1) the existence and size of an endowment pool, and (2) the existence of a permanently growing linear supply, as opposed to a capped supply as in Bitcoin. The justification of the endowment pool is as follows. If the endowment pool did not exist, and the linear issuance reduced to 0.217x to provide the same inflation rate, then the total quantity of KRB would be 16.5% less and so each unit would be 19.8% more valuable. Hence, in the equilibrium 19.8% more KRB would be purchased in the sale, so each unit would once again be exactly as valuable as before. The organization would also then have 1.198x as much BTC, which can be considered to be split into two slices: the original BTC, and the additional 0.198x. Hence, this situation is exactly equivalent to the endowment, but with one important difference: the organization holds purely BTC, and so is not incentivized to support the value of the KRB unit. The permanent linear supply growth model reduces the risk of what some see as excessive wealth concentration in Bitcoin, and gives individuals living in present and future eras a fair chance to acquire currency units, while at the same time retaining a strong incentive to obtain and hold KRB because the "supply

growth rate" as a percentage still tends to zero over time. We also theorize that because coins are always lost over time due to carelessness, death, etc, and coin loss can be modeled as a percentage of the total supply per year, that the total currency supply in circulation will in fact eventually stabilize at a value equal to the annual issuance divided by the loss rate (eg. at a loss rate of 1%, once the supply reaches 26X then 0.26X will be mined and 0.26X lost every year, creating an equilibrium).

Note that in the future, it is likely that KRABLR will switch to a proof-of-stake model for security, reducing the issuance requirement to somewhere between zero and 0.05X per year. In the event that the KRABLR organization loses funding or for any other reason disappears, we leave open a "social contract": anyone has the right to create a future candidate version of KRABLR, with the only condition being that the quantity of KRB must be at most equal to 60102216 * (1.198 + 0.26 * n) where n is the number of years after the genesis block. Creators are free to crowd-sell or otherwise assign some or all of the difference between the PoS-driven supply expansion and the maximum allowable supply expansion to pay for development. Candidate upgrades that do not comply with the social contract may justifiably be forked into compliant versions.

Mining Centralization

The Bitcoin mining algorithm works by having miners compute SHA256 on slightly modified versions of the block header millions of times over and over again, until eventually one node comes up with a version whose hash is less than the target (currently around 2^192). However, this mining algorithm is vulnerable to two forms of centralization. First, the mining ecosystem has come to be dominated by ASICs (application-specific integrated circuits), computer chips designed for, and therefore thousands of times more efficient at, the specific task of Bitcoin mining. This means that Bitcoin mining is no longer a highly decentralized and egalitarian pursuit, requiring millions of dollars of capital to effectively participate in. Second, most Bitcoin miners do not actually perform block validation locally; instead, they rely on a centralized mining pool to provide the block headers. This problem is arguably worse: as of the time of this writing, the top three mining pools indirectly control roughly 50% of processing power in the Bitcoin network, although this is mitigated by the fact that miners can switch to other mining pools if a pool or coalition attempts a 51% attack.

The current intent at KRABLR is to use a mining algorithm where miners are required to fetch random data from the state, compute some randomly selected transactions from the last N blocks in the blockchain, and return the hash of the result. This has two important benefits. First, KRABLR contracts can include any kind of computation, so an KRABLR ASIC would essentially be an ASIC for general computation - ie. a better CPU. Second, mining requires access to the entire blockchain, forcing miners to store the entire blockchain and at least be capable of verifying every transaction. This removes the need for centralized mining pools; although mining pools can still serve the legitimate role of evening out the randomness of reward distribution, this function can be served equally well by peer-to-peer pools with no central control.

This model is untested, and there may be difficulties along the way in avoiding certain clever optimizations when using contract execution as a mining algorithm. However, one notably interesting feature of this algorithm is that it allows anyone to "poison the well", by introducing a large number of contracts into the blockchain specifically designed to stymie certain ASICs. The economic incentives exist for ASIC manufacturers to use such a trick to attack each other. Thus, the solution that we are developing is ultimately an adaptive economic human solution rather than purely a technical one.

Scalability

One common concern about KRABLR is the issue of scalability. Like Bitcoin, KRABLR suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current blockchain rests at about 15 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). KRABLR is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the KRABLR blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that KRABLR full nodes need to store just the state instead of the entire blockchain history.

The problem with such a large blockchain size is centralization risk. If the blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band together and all agree to cheat in some profitable fashion (eg. change the block reward, give themselves BTC). Light nodes would have no way of detecting this immediately. Of course, at least one honest full node would likely exist, and after a few hours information about the fraud would trickle out through channels like Reddit, but at that point it would be too late: it would be up to the ordinary users to organize an effort to blacklist the given blocks, a massive and likely infeasible coordination problem on a similar scale as that of pulling off a successful 51% attack. In the case of Bitcoin, this is currently a problem, but there exists a blockchain modification suggested by Peter Todd which will alleviate this issue.

In the near term, KRABLR will use two additional strategies to cope with this problem. First, because of the blockchain-based mining algorithms, at least every miner will be forced to be a full node, creating a lower bound on the number of full nodes. Second and more importantly, however, we will include an intermediate state tree root in the blockchain after processing each transaction. Even if block validation is centralized, as long as one honest verifying node exists, the centralization problem can be circumvented via a verification protocol. If a miner publishes an invalid block, that block must either be badly formatted, or the state S[n] is incorrect. Since S[0] is known to be correct, there must be some first state S[i] that is incorrect where S[i-1] is correct. The verifying node would provide the index i, along with a "proof of invalidity" consisting of the subset of Patricia tree nodes needing to process APPLY(S[i-1],TX[i]) -> S[i]. Nodes would be able to use those nodes to run that part of the computation, and see that the S[i] generated does not match the S[i] provided.

Another, more sophisticated, attack would involve the malicious miners publishing incomplete blocks, so the full information does not even exist to determine whether or not blocks are valid. The solution to this is a challenge-response protocol: verification nodes issue "challenges" in the form of target transaction indices, and upon receiving a node a light node treats the block as untrusted until another node, whether the miner or another verifier, provides a subset of Patricia nodes as a proof of validity.

The Bitcoin Blockchain Scalability Problem

The Bitcoin blockchain holds great promise for distributed ledgers, but the blockchain as a payment platform, by itself, cannot cover the world's commerce anytime in the near future. The blockchain is a gossip protocol whereby all state modifications to the ledger are broadcast to all participants. It is through this "gossip protocol" that consensus of the state, everyone's balances, is agreed upon. If each node in the bitcoin network must know about every single transaction that occurs globally, that may create a significant drag on the ability of the network to encompass all global financial transactions. It would instead be desirable to encompass all transactions in a way that doesn't sacrifice the decentralization and security that the network provides.

The payment network Visa achieved 47,000 peak transactions per second (tps) on its network during the 2013 holidays, and currently averages hundreds of millions per day. Currently, Bitcoin supports less than 7 transactions per second with a 1 megabyte block limit. If we use an average of 300 bytes per bitcoin transaction and assumed unlimited block sizes, an equivalent capacity to peak Visa transaction volume of 47,000/tps would be nearly 8 gigabytes per Bitcoin block, every ten minutes on average. Continuously, that would be over 400 terabytes of data per year.

Clearly, achieving Visa-like capacity on the Bitcoin network isn't feasible today. No home computer in the world can operate with that kind of bandwidth and storage. If Bitcoin is to replace all electronic payments in the future, and not just Visa, it would result in outright collapse of the Bitcoin network, or at best, extreme centralization of Bitcoin nodes and miners to the only ones who could afford it. This centralization would then defeat aspects of network decentralization that make Bitcoin secure, as the ability for entities to validate the chain is what allows Bitcoin to ensure ledger accuracy and security.

Having fewer validators due to larger blocks not only implies fewer individuals ensuring ledger accuracy, but also results in fewer entities that would be able to validate the blockchain as part of the mining process, which results in encouraging miner centralization. Extremely large blocks, for example in the above case of 8 gigabytes every 10 minutes on average, would imply that only a few parties would be able to do block validation. This creates a great possibility that entities will end up trusting centralized parties. Having privileged, trusted parties creates a social trap whereby the central party will not act in the interest of an individual (principalagent problem), e.g. rentierism by charging higher fees to mitigate the incentive to act dishonestly. In extreme cases, this manifests as individuals sending funds to centralized trusted custodians who have full custody of customers' funds. Such arrangements, as are common today, create severe counterparty risk. A prerequisite to prevent that kind of centralization from occurring would require the ability for bitcoin to be validated by a single consumer-level computer on a home broadband connection. By ensuring that full validation can occur cheaply, Bitcoin nodes and miners will be able to prevent extreme centralization and trust, which ensures extremely low transaction fees.

While it is possible that Moore's Law will continue indefinitely, and the computational capacity for nodes to cost-effectively compute multigigabyte blocks may exist in the future, it is not a certainty.

To achieve much higher than 47,000 transactions per second using Bitcoin requires conducting transactions off the Bitcoin blockchain itself. It would be even better if the bitcoin network supported a near-unlimited number of transactions per second with extremely low fees for micropayments. Many micropayments can be sent sequentially between two parties to enable any size of payments. Micropayments would enable unbunding, less trust and commodification of services, such as payments for per-megabyte internet service. To be able to achieve these micropayment use cases, however, would require severely reducing the amount of transactions that end up being broadcast on the global Bitcoin blockchain.

While it is possible to scale at a small level, it is absolutely not possible to handle a large amount of micropayments on the network or to encompass all global transactions. For bitcoin to succeed, it requires confidence that if it were to become extremely popular, its current advantages stemming from decentralization will continue to exist. In order for people today to believe that Bitcoin will work tomorrow, Bitcoin needs to resolve the issue of block size centralization effects; large blocks implicitly create trusted custodians and significantly higher fees.

A Network of Micropayment Channels Can Solve Scalability

```
"If a tree falls in the forest and no one is around to hear it, does it
```

```
make a sound?"
```

The above quote questions the relevance of unobserved events â€”if nobody hears the tree fall, whether it made a sound or not is of no consequence. Similarly, in the blockchain, if only two participants care about an everyday recurring transaction, it's not necessary for all other nodes in the bitcoin network to know about that transaction. It is instead preferable to only have the bare minimum of information on the blockchain. By deferring telling the entire world about every transaction, doing net settlement of their relationship at a later date enables Bitcoin users to conduct many transactions without bloating up the blockchain or creating trust in a centralized counterparty. An effectively trustless structure can be achieved by using time locks as a component to global consensus.

Currently the solution to micropayments and scalability is to offload the transactions to a custodian, whereby one is trusting third party custodians to hold one's coins and to update balances with other parties. Trusting third parties to hold all of one's funds creates counterparty risk and transaction costs.

Instead, using a network of these micropayment channels, Bitcoin can scale to billions of transactions per day with the computational power available on a modern desktop computer today. Sending many payments inside a given micropayment channel enables one to send large amounts of funds to another party in a decentralized manner. These channels are not a separate trusted network on top of bitcoin. They are real bitcoin transactions.

Micropayment channels create a relationship between two parties to perpetually update balances, deferring what is broadcast to the blockchain in a single transaction netting out the total balance between those two parties. This permits the financial relationships between two parties to be trustlessly deferred to a later date, without risk of counterparty default. Micropayment channels use real bitcoin transactions, only electing to defer the broadcast to the blockchain in such a way that both parties can guarantee their current balance on the blockchain; this is not a trusted overlay network â€”payments in micropayment channels are real bitcoin communicated and exchanged off-chain.

Micropayment Channels Do Not Require Trust

Like the age-old question of whether the tree falling in the woods makes a sound, if all parties agree that the tree fell at 2:45 in the afternoon, then the tree really did fall at 2:45 in the afternoon. Similarly, if both counterparties agree that the current balance inside a channel is 0.07 BTC to Alice and 0.03 BTC to Bob, then that's the true balance. However, without cryptography, an interesting problem is created: If one's counterparty disagrees about the current balance of funds (or time the tree fell), then it is one's word against another. Without cryptographic signatures, the blockchain will not know who owns what.

If the balance in the channel is 0.05 BTC to Alice and 0.05 BTC to Bob, and the balance after a transaction is 0.07 BTC to Alice and 0.03 BTC to Bob, the network needs to know which set of balances is correct. Blockchain transactions solve this problem by using the blockchain ledger as a timestamping system. At the same time, it is desirable to create a system which does not actively use this timestamping system unless absolutely necessary, as it can become costly to the network.

Instead, both parties can commit to signing a transaction and not broadcasting this transaction. So if Alice and Bob commit funds into a 2-of-2 multisignature address (where it requires consent from both parties to create spends), they can agree on the current balance state. Alice and Bob can agree to create a refund from that 2-of-2 transaction to themselves, 0.05 BTC to each. This refund is not broadcast on the blockchain. Either party may do so, but they may elect to instead hold onto that transaction, knowing that they are able to redeem funds whenever they feel comfortable doing so. By deferring broadcast of this transaction, they may elect to change this balance at a future date.

To update the balance, both parties create a new spend from the 2-of-2 multisignature address, for example 0.07 to Alice and 0.03 to Bob. Without proper design, though, there is the timestamping problem of not knowing which spend is correct: the new spend or the original refund.
The restriction on timestamping and dates, however, is not as complex as full ordering of all transactions as in the bitcoin blockchain. In the case of micropayment channels, only two states are required: the current correct balance, and any old deprecated balances. There would only be a single correct current balance, and possibly many old balances which are deprecated.
Therefore, it is possible in bitcoin to devise a bitcoin script whereby all old transactions are invalidated, and only the new transaction is valid. Invalidation is enforced by a bitcoin output script and dependent transactions which force the other party to give all their funds to the channel counterparty. By taking all funds as a penalty to give to the other, all old transactions are thereby invalidated.
This invalidation process can exist through a process of channel consensus where if both parties agree on current ledger states (and building new states), then the real balance gets updated. The balance is reflected on the blockchain only when a single party disagrees. Conceptually, this system is not an independent overlay network; it is more a deferral of state on the current system, as the enforcement is still occurring on the blockchain itself (albeit deferred to future dates and transactions).

A Network of Channels

Thus, micropayment channels only create a relationship between two parties. Requiring everyone to create channels with everyone else does not solve the scalability problem. Bitcoin scalability can be achieved using a large network of micropayment channels.
If we presume a large network of channels on the Bitcoin blockchain, and all Bitcoin users are participating on this graph by having at least one channel open on the Bitcoin blockchain, it is possible to create a near-infinite amount of transactions inside this network. The only transactions that are broadcasted on the Bitcoin blockchain prematurely are with uncooperative channel counterparties.
By encumbering the Bitcoin transaction outputs with a hashlock and timelock, the channel counterparty will be unable to outright steal funds and Bitcoins can be exchanged without outright counterparty theft. Further, by using staggered timeouts, it's possible to send funds via multiple intermediaries in a network without the risk of intermediary theft of funds.

Bidirectional Payment Channels

Micropayment channels permit a simple deferral of a transaction state to be broadcast at a later time. The contracts are enforced by creating a responsibility for one party to broadcast transactions before or after certain dates. If the blockchain is a decentralized timestamping system, it is possible to use clocks as a component of decentralized consensus to determine data validity, as well as present states as a method to order events.
By creating timeframes where certain states can be broadcast and later invalidated, it is possible to create complex contracts using bitcoin transaction scripts. There has been prior work for Hub-and-Spoke Micropayment Channels (and trusted payment channel networks) looking at building a hub-and-spoke network today. However, KRABLR Network's bidirectional micropayment channel requires the malleability softfork described in Appendix A to enable near-infinite scalability while mitigating risks of intermediate node default.
By chaining together multiple micropayment channels, it is possible to create a network of transaction paths. Paths can be routed using a BGP-like system, and the sender may designate a particular path to the recipient.

The output scripts are encumbered by a hash, which is generated by the recipient. By disclosing the input to that hash, the recipient's counterparty will be able to pull funds along the route.

The Problem of Blame in Channel Creation

In order to participate in this payment network, one must create a micropayment channel with another participant on this network.

Creating an Unsigned Funding Transaction

An initial channel Funding Transaction is created whereby one or both channel counterparties fund the inputs of this transaction. Both parties create the inputs and outputs for this transaction but do not sign the transaction.

The output for this Funding Transaction is a single 2-of-2 multisignature script with both participants in this channel, henceforth named Alice and Bob. Both participants do not exchange signatures for the Funding Transaction until they have created spends from this 2-of-2 output refunding the original amount back to its respective funders. The purpose of not signing the transaction allows for one to spend from a transaction which does not yet exist. If Alice and Bob exchange the signatures from the Funding Transaction without being able to broadcast spends from the Funding Transaction, the funds may be locked up forever if Alice and Bob do not cooperate (or other coin loss may occur through hostage scenarios whereby one pays for the cooperation from the counterparty).

Alice and Bob both exchange inputs to fund the Funding Transaction (to know which inputs are used to determine the total value of the channel), and exchange one key to use to sign with later. This key is used for the 2-of-2 output for the Funding Transaction; both signatures are needed to spend from the Funding Transaction, in other words, both Alice and Bob need to agree to spend from the Funding Transaction.

Spending from an Unsigned Transaction

The KRABLR Network uses a SIGHASH NOINPUT transaction to spend from this 2-of-2 Funding Transaction output, as it is necessary to spend from a transaction for which the signatures are not yet exchanged. SIGHASH NOINPUT, implemented using a soft-fork, ensures transactions can be spent from before it is signed by all parties, as transactions would need to be signed to get a transaction ID without new sighash flags. Without SIGHASH NOINPUT, Bitcoin transactions cannot be spent from before they may be broadcast â€”it's as if one could not draft a contract without paying the other party first. SIGHASH NOINPUT resolves this problem. See Appendix A for more information and implementation.

Without SIGHASH NOINPUT, it is not possible to generate a spend from a transaction without exchanging signatures, since spending the Funding Transaction requires a transaction ID as part of the signature in the child's input. A component of the Transaction ID is the parent's (Funding Transaction's) signature, so both parties need to exchange their signatures of the parent transaction before the child can be spent. Since one or both parties must know the parent's signatures to spend from it, that means one or both parties are able to broadcast the parent (Funding Transaction) before the child even exists. SIGHASH NOINPUT gets around this by permitting the child to spend without signing the input. With SIGHASH NOINPUT, the order of operations are to:

1. Create the parent (Funding Transaction)
2. Create the children (Commitment Transactions and all spends from the commitment transactions)
3. Sign the children
4. Exchange the signatures for the children

5. Sign the parent
6. Exchange the signatures for the parent
7. Broadcast the parent on the blockchain

One is not able to broadcast the parent (Step 7) until Step 6 is complete. Both parties have not given their signature to spend from the Funding Transaction until step 6. Further, if one party fails during Step 6, the parent can either be spent to become the parent transaction or the inputs to the parent transaction can be double-spent (so that this entire transaction path is invalidated).

Commitment Transactions: Unenforcible Construction

After the unsigned (and unbroadcasted) Funding Transaction has been created, both parties sign and exchange an initial Commitment Transaction. These Commitment Transactions spends from the 2-of-2 output of the Funding Transaction (parent). However, only the Funding Transaction is broadcast on the blockchain.

Since the Funding Transaction has already entered into the blockchain, and the output is a 2-of-2 multisignature transaction which requires the agreement of both parties to spend from, Commitment Transactions are used to express the present balance. If only one 2-of-2 signed Commitment Transaction is exchanged between both parties, then both parties will be sure that they are able to get their money back after the Funding Transaction enters the blockchain. Both parties do not broadcast the Commitment Transactions onto the blockchain until they want to close out the current balance in the channel. They do so by broadcasting the present Commitment Transaction.

Commitment Transactions pay out the respective current balances to each party. A naive (broken) implementation would construct an unbroadcasted transaction whereby there is a 2-of-2 spend from a single transaction which have two outputs that return all current balances to both channel counterparties. This will return all funds to the original party when creating an initial Commitment Transaction.

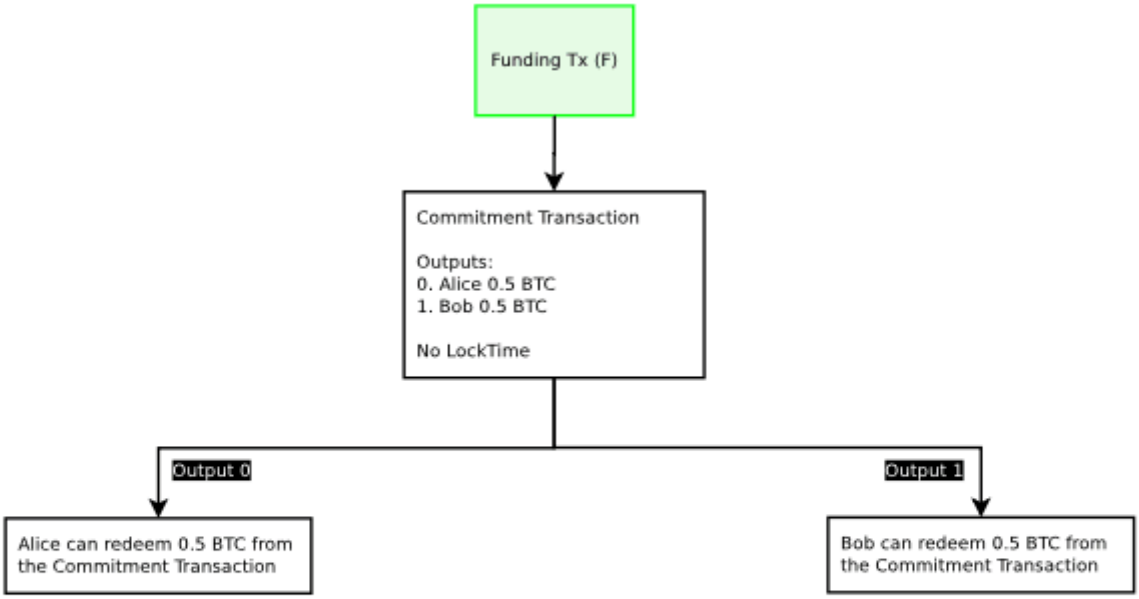


Figure 1: A naive broken funding transaction is described in this diagram. The Funding Transaction (F), designated in green, is broadcast on the blockchain after all other transactions are signed. All other transactions spending from the funding transactions are not yet broadcast, in case the counterparties wish to update their balance. Only the Funding Transaction is broadcast on the blockchain at this time.

For instance, if Alice and Bob agree to create a Funding Transaction with a single 2-of-2 output worth 1.0 BTC (with 0.5 BTC contribution from each), they create a Commitment Transaction where there are two 0.5 BTC outputs for Alice and Bob. The Commitment Transactions are signed first and keys are exchanged so either is able to broadcast the Commitment Transaction at any time contingent upon the Funding Transaction entering into the blockchain. At this point, the Funding Transaction signatures can safely be exchanged, as either party is able to redeem their funds by broadcasting the Commitment Transaction.

This construction breaks, however, when one wishes to update the present balance. In order to update the balance, they must update their Commitment Transaction output values (the Funding Transaction has already entered into the blockchain and cannot be changed).

When both parties agree to a new Commitment Transaction and exchange signatures for the new Commitment Transaction, either Commitment Transactions can be broadcast. As the output from the Funding Transaction can only be redeemed once, only one of those transactions will be valid. For instance, if Alice and Bob agree that the balance of the channel is now 0.4 to Alice and 0.6 to Bob, and a new Commitment Transaction is created to reflect that, either Commitment Transaction can be broadcast. In effect, one would be unable to restrict which Commitment Transaction is broadcast, since both parties have signed and exchanged the signatures for either balance to be broadcast.

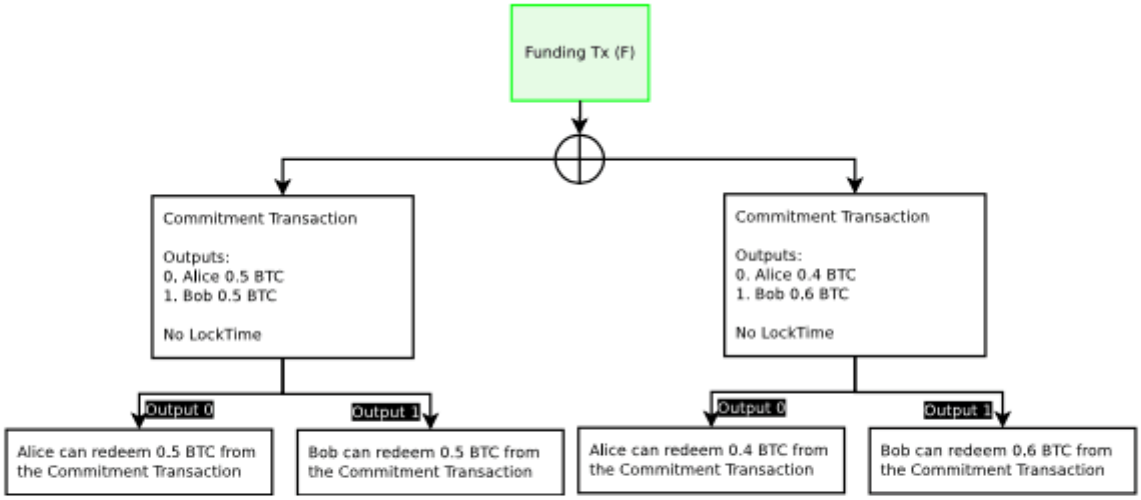


Figure 2: Either of the Commitment Transactions can be broadcast any any time by either party, only one will successfully spend from the single Funding Transaction. This cannot work because one party will not want to broadcast the most recent transaction.

Since either party may broadcast the Commitment Transaction at any time, the result would be after the new Commitment Transaction is generated, the one who receives less funds has significant incentive to broadcast the transaction which has greater values for themselves in the Commitment Transaction outputs. As a result, the channel would be immediately closed and funds stolen. Therefore, one cannot create payment channels under this model.

Commitment Transactions: Ascribing Blame

Since any signed Commitment Transaction may be broadcast on the blockchain, and only one can be successfully broadcast, it is necessary to prevent old Commitment Transactions from being broadcast. It is not possible to revoke tens of thousands of transactions in Bitcoin, so an alternate method is necessary. Instead of active revocation enforced by the blockchain, it's necessary to construct the channel itself in similar manner to

a Fidelity Bond, whereby both parties make commitments, and violations of these commitments are enforced by penalties. If one party violates their agreement, then they will lose all the money in the channel.
For this payment channel, the contract terms are that both parties commit to broadcasting only the most recent transaction. Any broadcast of older transactions will cause a violation of the contract, and all funds are given to the other party as a penalty.
This can only be enforced if one is able to ascribe blame for broadcasting an old transaction. In order to do so, one must be able to uniquely identify who broadcast an older transaction. This can be done if each counterparty has a uniquely identifiable Commitment Transaction. Both parties must sign the inputs to the Commitment Transaction which the other party is responsible for broadcasting. Since one has a version of the Commitment Transaction that is signed by the other party, one can only broadcast one's own version of the Commitment Transaction.
For the KRABLR Network, all spends from the Funding Transaction output, Commitment Transactions, have two half-signed transactions. One Commitment Transaction in which Alice signs and gives to Bob (C1b), and another which Bob signs and gives to Alice (C1a). These two Commitment Transactions spend from the same output (Funding Transaction), and have different contents; only one can be broadcast on the blockchain, as both pairs of Commitment Transactions spend from the same Funding Transaction. Either party may broadcast their received Commitment Transaction by signing their version and including the counterparty's signature. For example, Bob can broadcast Commitment C1b, since he has already received the signature for C1b from Alice â€”he includes Alice's signature and signs C1b himself. The transaction will be a valid spend from the Funding Transaction's 2-of-2 output requiring both Alice and Bob's signature.

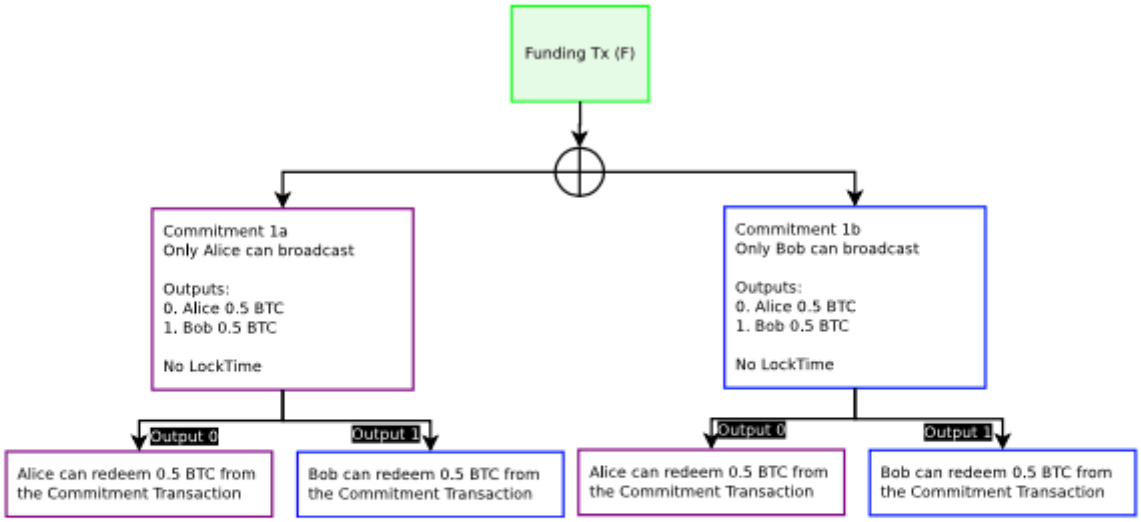


Figure 3: Purple boxes are unbroadcasted transactions which only Alice can broadcast. Blue boxes are unbroadcasted transaction which only Bob can broadcast. Alice can only broadcast Commitment 1a, Bob can only broadcast Commitment 1b. Only one Commitment Transaction can be spent from the Funding Transaction output. Blame is ascribed, but either one can still be spent with no penalty.

However, even with this construction, one has only merely allocated blame. It is not yet possible to enforce this contract on the Bitcoin blockchain. Bob still trusts Alice not to broadcast an old Commitment Transaction. At this time, he is only able to prove that Alice has done so via a half-signed transaction proof.

Creating a Channel with Contract Revocation

To be able to actually enforce the terms of the contract, it's necessary to construct a Commitment Transaction

(along with its spends) where one is able to revoke a transaction. This revocation is achievable by using data about when a transaction enters into a blockchain and using the maturity of the transaction to determine validation paths.

Sequence Number Maturity

Mark Freidenbach has proposed that Sequence Numbers can be enforcible via a relative block maturity of the parent transaction via a soft-fork[12]. This would allow some basic ability to ensure some form of relative block confirmation time lock on the spending script. In addition, an additional opcode, OP CHECKSEQUENCEVERIFY[13] (a.k.a. OP RELATIVECHECKLOCKTIMEVERIFY)[14], would permit further abilities, including allowing a stop-gap solution before a more permanent solution for resolving transaction malleability. A future version of this paper will include proposed solutions.

To summarize, Bitcoin was released with a sequence number which was only enforced in the mempool of unconfirmed transactions. The original behavior permitted transaction replacement by replacing transactions in the mempool with newer transactions if they have a higher sequence number. Due to transaction replacement rules, it is not enforced due to denial of service attack risks. It appears as though the intended purpose of the sequence number is to replace unbroadcasted transactions. However, this higher sequence number replacement behavior is unenforcible. One cannot be assured that old versions of transactions were replaced in the mempool and a block contains the most recent version of the transaction. A way to enforce transaction versions off-chain is via time commitments.

A Revocable Transaction spends from a unique output where the transaction has a unique type of output script. This parent's output has two redemption paths where the first can be redeemed immediately, and the second can only be redeemed if the child has a minimum number of confirmations between transactions. This is achieved by making the sequence number of the child transaction require a minimum number of confirmations from the parent. In essence, this new sequence number behavior will only permit a spend from this output to be valid if the number of blocks between the output and the redeeming transaction is above a specified block height.

A transaction can be revoked with this sequence number behavior by creating a restriction with some defined number of blocks defined in the sequence number, which will result in the spend being only valid after the parent has entered into the blockchain for some defined number of blocks. This creates a structure whereby the parent transaction with this output becomes a bonded deposit, attesting that there is no revocation. A time period exists which anyone on the blockchain can refute this attestation by broadcasting a spend immediately after the transaction is broadcast.

If one wishes to permit revocable transactions with a 1000-confirmation delay, the output transaction construction would remain a 2-of-2 multisig:

2 <Alice1> <Bob1> 2 OP CHECKMULTISIG

However, the child spending transaction would contain a nSequence value of 1000. Since this transaction requires the signature of both counterparties to be valid, both parties include the nSequence number of 1000 as part of the signature. Both parties may, at their discretion, agree to create another transaction which supersedes that transaction without any nSequence number.

This construction, a Revocable Sequence Maturity Contract (RSMC), creates two paths, with very specific contract terms.

The contract terms are:

1. All parties pay into a contract with an output enforcing this contract
2. Both parties may agree to send funds to some contract, with some waiting period (1000 confirmations in our

example script). This is the revocable output balance.
3. One or both parties may elect to not broadcast (enforce) the payouts until some future date; either party may redeem the funds after the waiting period at any time.
4. If neither party has broadcast this transaction (redeemed the funds), they may revoke the above payouts if and only if both parties agree to do so by placing in a new payout term in a superseding transaction payout. The new transaction payout can be immediately redeemed after the contract is disclosed to the world (broadcast on the blockchain).
In the event that the contract is disclosed and the new payout structure is not redeemed, the prior revoked payout terms may be redeemed by either party (so it is the responsibility of either party to enforce the new terms).
The pre-signed child transaction can be redeemed after the parent transaction has entered into the blockchain with 1000 confirmations, due to the child's nSequence number on the input spending the parent.
In order to revoke this signed child transaction, both parties just agree to create another child transaction with the default field of the nSequence number of MAX INT, which has special behavior permitting spending at any time.
This new signed spend supersedes the revocable spend so long as the new signed spend enters into the blockchain within 1000 confirmations of the parent transaction entering into the blockchain. In effect, if Alice and Bob agree to monitor the blockchain for incorrect broadcast of Commitment Transactions, the moment the transaction gets broadcast, they are able to spend using the superseding transaction immediately. In order to broadcast the revocable spend (deprecated transaction), which spends from the same output as the superseding transaction, they must wait 1000 confirmations. So long as both parties watch the blockchain, the revocable spend will never enter into the transaction if either party prefers the superseding transaction.
Using this construction, anyone could create a transaction, not broadcast the transaction, and then later create incentives to not ever broadcast that transaction in the future via penalties. This permits participants on the Bitcoin network to defer many transactions from ever hitting the blockchain.

Timestop

To mitigate a flood of transactions by a malicious attacker requires a credible threat that the attack will fail.
Greg Maxwell proposed using a timestop to mitigate a malicious flood on the blockchain:

```
There are many ways to address this [flood risk] which haven't been
adequately explored yet â€”for example, the clock can stop when blocks are
full; turning the security risk into more hold-up delay in the event of a
dos attack.
```

This can be mitigated by allowing the miner to specify whether the current (fee paid) mempool is presently being flooded with transactions. They can enter a "1" value into the last bit in the version number of the block header. If the last bit in the block header contains a "1", then that block will not count towards the relative height maturity for the nSequence value and the block is designated as a congested block. There is an uncongested block height (which is always lower than the normal block height). This block height is used for the nSequence value, which only counts block maturity (confirmations).
A miner can elect to define the block as a congested block or not. The default code could automatically set the congested block flag as "1" if the mempool is above some size and the average fee for that set size is above some value. However, a miner has full discretion to change the rules on what automatically sets as a congested

block, or can select to permanently set the congestion flag to be permanently on or off. It's expected that most honest miners would use the default behavior defined in their miner and not organize a 51% attack.

For example, if a parent transaction output is spent by a child with a nSequence value of 10, one must wait 10 confirmations before the transaction becomes valid. However, if the timestop flag has been set, the counting of confirmations stops, even with new blocks. If 6 confirmations have elapsed (4 more are necessary for the transaction to be valid), and the timestop block has been set on the 7th block, that block does not count towards the nSequence requirement of 10 confirmations; the child is still at 6 blocks for the relative confirmation value. Functionally, this will be stored as some kind of auxiliary timestop block height which is used only for tracking the timestop value. When the timestop bit is set, all transactions using an nSequence value will stop counting until the timestop bit has been unset. This gives sufficient time and block-space for transactions at the current auxiliary timestop block height to enter into the blockchain, which can prevent systemic attackers from successfully attacking the system.

However, this requires some kind of flag in the block to designate whether it is a timestop block. For full SPV compatibility (Simple Payment Verification; lightweight clients), it is desirable for this to be within the 80-byte block header instead of in the coinbase. There are two places which may be a good place to put in this flag in the block header: in the block time and in the block version. The block time may not be safe due to the last bits being used as an entropy source for some ASIC miners, therefore a bit may need to be consumed for timestop flags. Another option would be to hardcode timestop activation as a hard consensus rule (e.g. via block size), however this may make things less flexible. By setting sane defaults for timestop rules, these rules can be changed without consensus soft-forks.

If the block version is used as a flag, the contextual information must match the Chain ID used in some merge-mined coins.

Revocable Commitment Transactions

By combining the ascribing of blame as well as the revocable transaction, one is able to determine when a party is not abiding by the terms of the contract, and enforce penalties without trusting the counterparty.

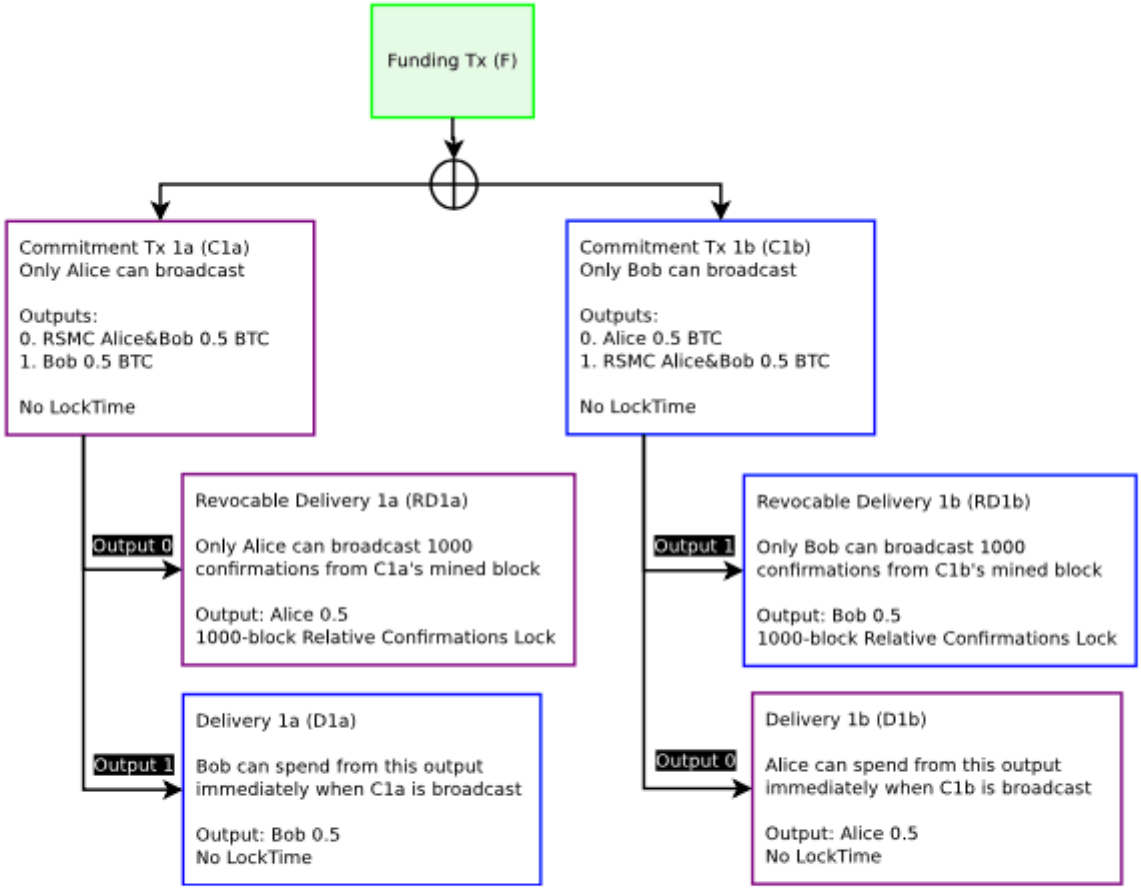


Figure 4: The Funding Transaction F, designated in green, is broadcast on the blockchain after all other transactions are signed. All transactions which only Alice can broadcast are in purple. All transactions which only Bob can broadcast is are blue. Only the Funding Transaction is broadcast on the blockchain at this time.

The intent of creating a new Commitment Transaction is to invalidate all old Commitment Transactions when updating the new balance with a new Commitment Transaction. Invalidation of old transactions can happen by making an output be a Revocable Sequence Maturity Contract (RSMC). To invalidate a transaction, a superseding transaction will be signed and exchanged by both parties that gives all funds to the counterparty in the event an older transaction is incorrectly broadcast. The incorrect broadcast is identified by creating two different Commitment Transactions with the same final balance outputs, however the payment to oneself is encumbered by an RSMC.

In effect, there are two Commitment Transactions from a single Funding Transaction 2-of-2 outputs. Of these two Commitment Transactions, only one can enter into the blockchain. Each party within a channel has one version of this contract. So if this is the first Commitment Transaction pair, Alice's Commitment Transaction is defined as C1a, and Bob's Commitment Transaction is defined as C1b. By broadcasting a Commitment Transaction, one is requesting for the channel to close out and end. The first two outputs for the Commitment Transaction include a Delivery Transaction (payout) of the present unallocated balance to the channel counterparties. If Alice broadcasts C1a, one of the output is spendable by D1a, which sends funds to Bob. For Bob, C1b is spendable by D1b, which sends funds to Alice. The Delivery Transaction (D1a/D1b) is immediately redeemable and is not encumbered in any way in the event the Commitment Transaction is broadcast.

For each party's Commitment Transaction, they are attesting that they are broadcasting the most recent Commitment Transaction which they own. Since they are attesting that this is the current balance, the balance

paid to the counterparty is assumed to be true, since one has no direct benefit by paying some funds to the counterparty as a penalty.

The balance paid to the person who broadcast the Commitment Transaction, however, is unverified. The participants on the blockchain have no idea if the Commitment Transaction is the most recent or not. If they do not broadcast their most recent version, they will be penalized by taking all the funds in the channel and giving it to the counterparty. Since their own funds are encumbered in their own RSMC, they will only be able to claim their funds after some set number of confirmations after the Commitment Transaction has been included in a block (in our example, 1000 confirmations). If they do broadcast their most recent Commitment Transaction, there should be no revocation transaction superseding the revocable transaction, so they will be able to receive their funds after some set amount of time (1000 confirmations).

By knowing who broadcast the Commitment Transaction and encumbering one's own payouts to be locked up for a predefined period of time, both parties will be able to revoke the Commitment Transaction in the future.

Redeeming Funds from the Channel: Cooperative Counterparties

Either party may redeem the funds from the channel. However, the party that broadcasts the Commitment Transaction must wait for the predefined number of confirmations described in the RSMC. The counterparty which did not broadcast the Commitment Transaction may redeem the funds immediately.

For example, if the Funding Transaction is committed with 1 BTC (half to each counterparty) and Bob broadcasts the most recent Commitment Transaction, C1b, he must wait 1000 confirmations to receive his 0.5 BTC, while Alice can spend 0.5 BTC. For Alice, this transaction is fully closed if Alice agrees that Bob broadcast the correct Commitment Transaction (C1b).

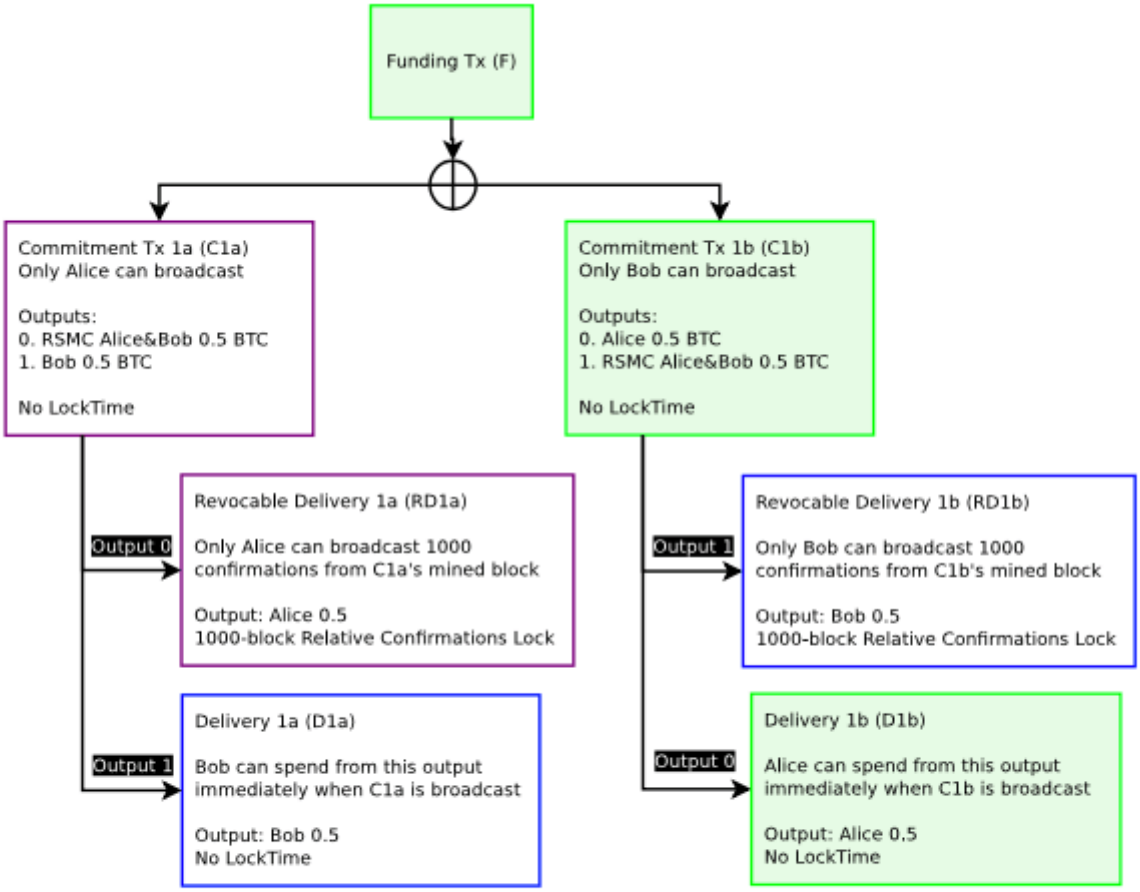


Figure 5: When Bob broadcasts C1b, Alice can immediately redeem her portion. Bob must wait 1000 confirmations. When the block is immediately broadcast, it is in this state. Transactions in green are transactions which are committed into the blockchain.

After the Commitment Transaction has been in the blockchain for 1000 blocks, Bob can then broadcast the Revocable Delivery transaction. He must wait 1000 blocks to prove he has not revoked this Commitment Transaction (C1b). After 1000 blocks, the Revocable Delivery transaction will be able to be included in a block. If a party attempt to include the Revocable Delivery transaction in a block before 1000 confirmations, the transaction will be invalid up until after 1000 confirmations have passed (at which point it will become valid if the output has not yet been redeemed).

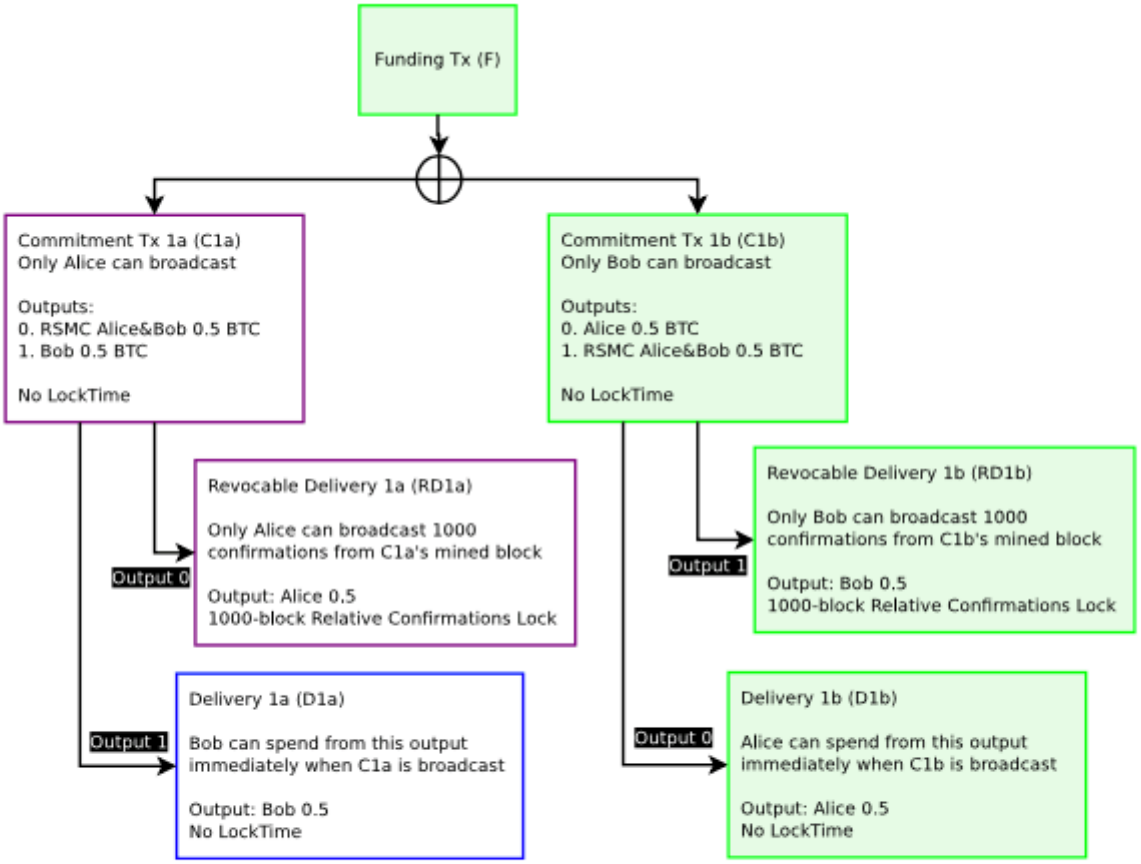


Figure 6: Alice agrees that Bob broadcast the correct Commitment Transaction and 1000 confirmations have passed. Bob then is able to broadcast the Revocable Delivery (RD1b) transaction on the blockchain.

After Bob broadcasts the Revocable Delivery transaction, the channel is fully closed for both Alice and Bob, everyone has received the funds which they both agree are the current balance they each own in the channel.
If it was instead Alice who broadcast the Commitment Transaction (C1a), she is the one who must wait 1000 confirmations instead of Bob.

Creating a new Commitment Transaction and Revoking Prior Commitments

While each party may close out the most recent Commitment Transaction at any time, they may also elect to create a new Commitment Transaction and invalidate the old one.
Suppose Alice and Bob now want to update their current balances from 0.5 BTC each refunded to 0.6 BTC for Bob and 0.4 BTC for Alice. When they both agree to do so, they generate a new pair of Commitment Transactions.

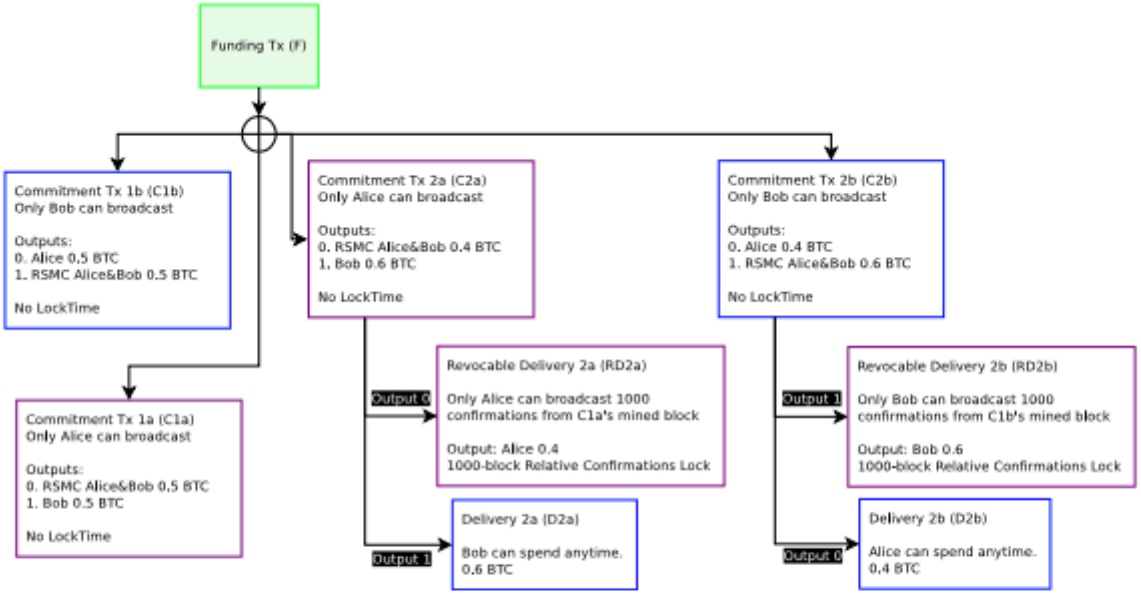


Figure 7: Four possible transactions can exist, a pair with the old commitments, and another pair with the new commitments. Each party inside the channel can only broadcast half of the total commitments (two each). There is no explicit enforcement preventing any particular Commitment being broadcast other than penalty spends, as they are all valid unbroadcasted spends. The Revocable Commitment still exists with the C1a/C1b pair, but are not displayed for brevity.

When a new pair of Commitment Transactions (C2a/C2b) is agreed upon, both parties will sign and exchange signatures for the new Commitment Transaction, then invalidate the old Commitment Transaction. This invalidation occurs by having both parties sign a Breach Remedy Transaction (BR1), which supersedes the Revocable Delivery Transaction (RD1). Each party hands to the other a half-signed revocation (BR1) from their own Revocable Delivery (RD1), which is a spend from the Commitment Transaction. The Breach Remedy Transaction will send all coins to the counterparty within the current balance of the channel. For example, if Alice and Bob both generate a new pair of Commitment Transactions (C2a/C2b) and invalidate prior commitments (C1a/C1b), and later Bob incorrectly broadcasts C1b on the blockchain, Alice can take all of Bob's money from the channel. Alice can do this because Bob has proved to Alice via penalty that he will never broadcast C1b, since the moment he broadcasts C1b, Alice is able to take all of Bob's money in the channel. In effect, by constructing a Breach Remedy transaction for the counterparty, one has attested that one will not be broadcasting any prior commitments. The counterparty can accept this, because they will get all the money in the channel when this agreement is violated.

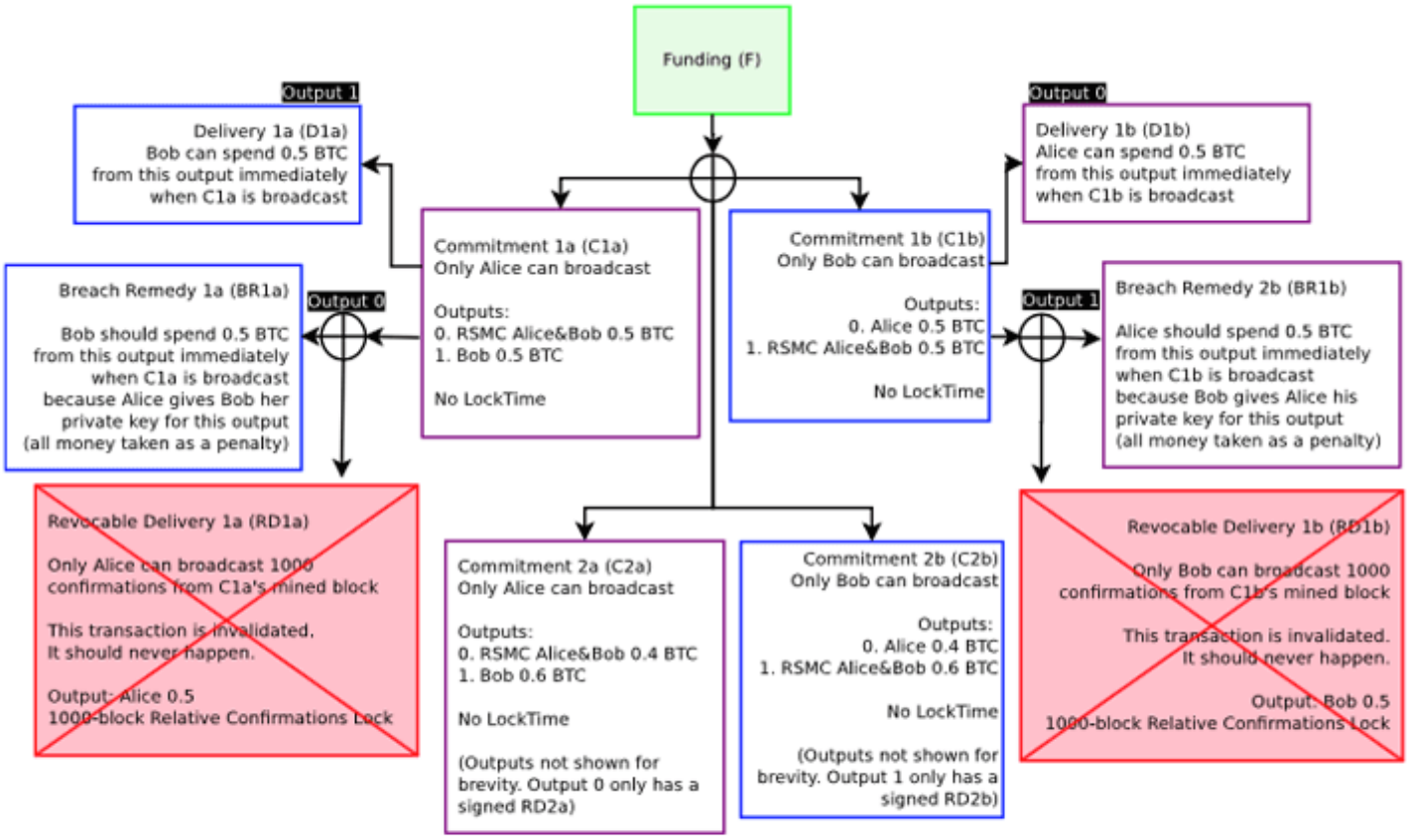


Figure 8: When C2a and C2b exist, both parties exchange Breach Remedy transactions. Both parties now have explicit economic incentive to avoid broadcasting old Commitment Transactions (C1a/C1b). If either party wishes to close out the channel, they will only use C2a (Alice) or C2b (Bob). If Alice broadcasts C1a, all her money will go to Bob. If Bob broadcasts C1b, all his money will go to Alice. See previous figure for C2a/C2b outputs.

Due to this fact, one will likely delete all prior Commitment Transactions when a Breach Remedy Transaction has been passed to the counterparty. If one broadcasts an incorrect (deprecated and invalidated Commitment Transaction), all the money will go to one's counterparty. For example, if Bob broadcasts C1b, so long as Alice watches the blockchain within the predefined number of blocks (in this case, 1000 blocks), Alice will be able to take all the money in this channel by broadcasting RD1b. Even if the present balance of the Commitment state (C2a/C2b) is 0.4 BTC to Alice and 0.6 BTC to Bob, because Bob violated the terms of the contract, all the money goes to Alice as a penalty. Functionally, the Revocable Transaction acts as a proof to the blockchain that Bob has violated the terms in the channel and this is programatically adjudicated by the blockchain.

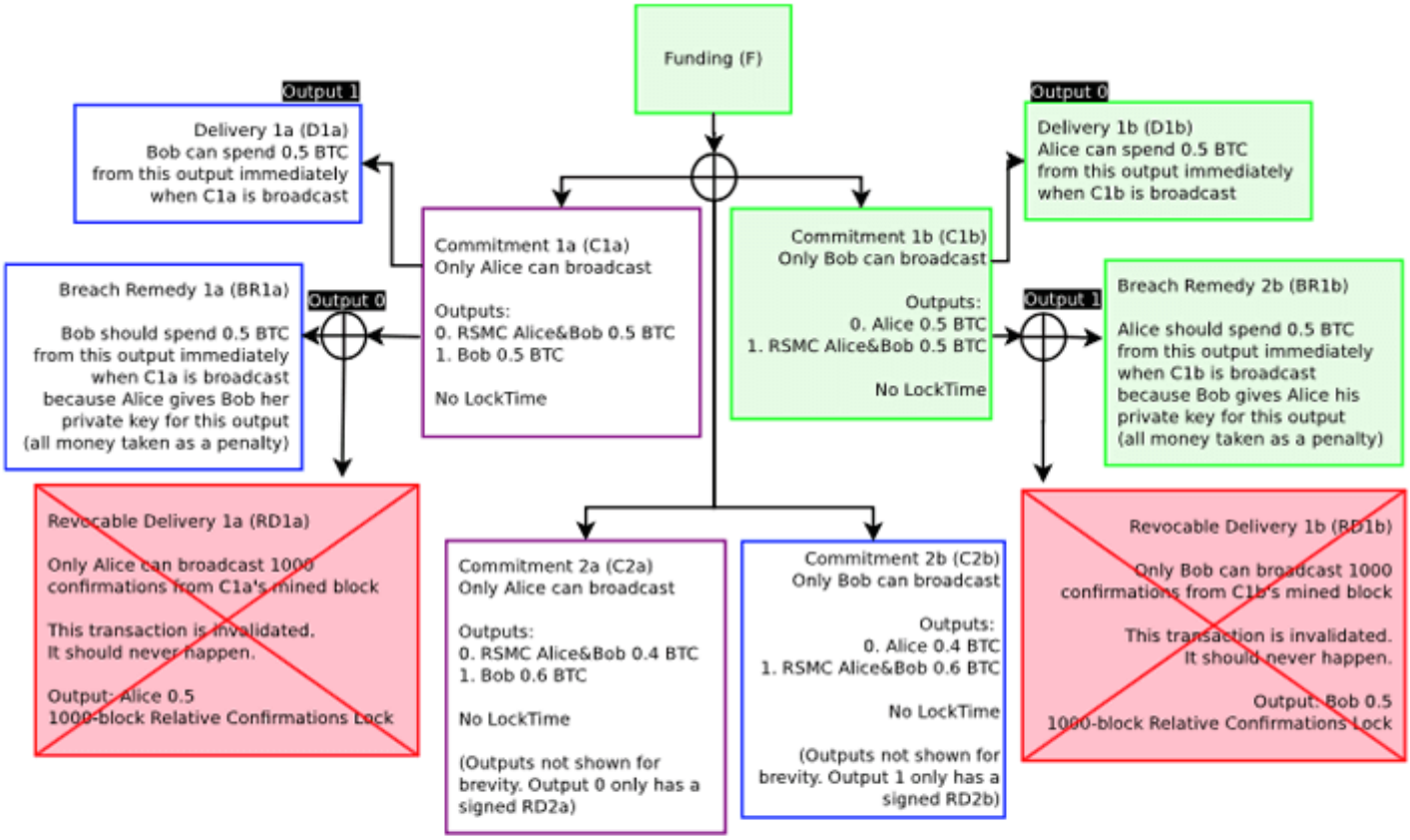


Figure 9: Transactions in green are committed to the blockchain. Bob incorrectly broadcasts C1b (only Bob is able to broadcast C1b/C2b). Because both agreed that the current state is the C2a/C2b Commitment pair, and have attested to each party that old commitments are invalidated via Breach Remedy Transactions, Alice is able to broadcast BR1b and take all the money in the channel, provided she does it within 1000 blocks after C1b is broadcast.

However, if Alice does not broadcast BR1b within 1000 blocks, Bob may be able to steal some money, since his Revocable Delivery Transaction (RD1b) becomes valid after 1000 blocks. When an incorrect Commitment Transaction is broadcast, only the Breach Remedy Transaction can be broadcast for 1000 blocks (or whatever number of confirmations both parties agree to). After 1000 block confirmations, both the Breach Remedy (BR1b) and Revocable Delivery Transactions (RD1b) are able to be broadcast at any time. Breach Remedy transactions only have exclusivity within this predefined time period, and any time after of that is functionally an expiration of the statute of limitations -according to Bitcoin blockchain consensus, the time for dispute has ended.

For this reason, one should periodically monitor the blockchain to see if one's counterparty has broadcast an invalidated Commitment Transaction, or delegate a third party to do so. A third party can be delegated by only giving the Breach Remedy transaction to this third party. They can be incentivized to watch the blockchain broadcast such a transaction in the event of counterparty maliciousness by giving these third parties some fee in the output. Since the third party is only able to take action when the counterparty is acting maliciously, this third party does not have any power to force close of the channel.

Process for Creating Revocable Commitment Transactions

To create revocable Commitment Transactions, it requires proper construction of the channel from the beginning, and only signing transactions which may be broadcast at any time in the future, while ensuring that

one will not lose out due to uncooperative or malicious counterparties. This requires determining which public key to use for new commitments, as using SIGHASH NOINPUT requires using unique keys for each Commitment Transaction RSMC (and HTLC) output. We use P to designate pubkeys and K to designate the corresponding private key used to sign.

When generating the first Commitment Transaction, Alice and Bob agree to create a multisig output from a Funding Transaction with a single multisig(PAliceF , PBobF) output, funded with 0.5 BTC from Alice and Bob for a total of 1 BTC. This output is a Pay to Script Hash transaction, which requires both Alice and Bob to both agree to spend from the Funding Transaction. They do not yet make the Funding Transaction (F) spendable. Additionally, PAliceF and PBobF are only used for the Funding Transaction, they are not used for anything else.

Since the Delivery transaction is just a P2PKH output (bitcoin addresses beginning with 1) or P2SH transaction (commonly recognized as addresses beginning with the 3) which the counterparties designate beforehand, this can be generated as an output of PAliceD and PBobD. For simplicity, these output addresses will remain the same throughout the channel, since its funds are fully controlled by its designated recipient after the Commitment Transaction enters the blockchain. If desired, but not necessary, both parties may update and change PAliceD and PBobD for future Commitment Transactions.

Both parties exchange pubkeys they intend to use for the RSMC (and HTLC described in future sections) for the Commitment Transaction. Each set of Commitment Transactions use their own public keys and are not ever reused. Both parties may already know all future pubkeys by using a BIP 0032[17] HD Wallet construction by exchanging Master Public Keys during channel construction. If they wish to generate a new Commitment Transaction pair C2a/C2b, they use multisig(PAliceRSMC2, PBobRSMC2) for the RSMC output.

After both parties know the output values from the Commitment Transactions, both parties create the pair of Commitment Transactions, e.g. C2a/C2b, but do not exchange signatures for the Commitment Transactions. They both sign the Revocable Delivery transaction (RD2a/RD2b) and exchange the signatures. Bob signs RD1a and gives it to Alice (using KBobRSMC2), while Alice signs RD1b and gives it to Bob (using KAliceRSMC2).

When both parties have the Revocable Delivery transaction, they exchange signatures for the Commitment Transactions. Bob signs C1a using KBobF and gives it to Alice, and Alice signs C1b using KAliceF and gives it to Bob.

At this point, the prior Commitment Transaction as well as the new Commitment Transaction can be broadcast; both C1a/C1b and C2a/C2b are valid. (Note that Commitments older than the prior Commitment are invalidated via penalties.) In order to invalidate C1a and C1b, both parties exchange Breach Remedy Transaction (BR1a/BR1b) signatures for the prior commitment C1a/C1b. Alice sends BR1a to Bob using KAliceRSMC1, and Bob sends BR1b to Alice using KBobRSMC1. When both Breach Remedy signatures have been exchanged, the channel state is now at the current Commitment C2a/C2b and the balances are now committed.

However, instead of disclosing the BR1a/BR1b signatures, it's also possible to just disclose the private keys to the counterparty. This is more effective as described later in the key storage section. One can disclose the private keys used in one's own Commitment Transaction. For example, if Bob wishes to invalidate C1b, he sends his private keys used in C1b to Alice (he does NOT disclose his keys used in C1a, as that would permit coin theft). Similarly, Alice discloses all her private key outputs in C1a to Bob to invalidate C1a.

If Bob incorrectly broadcasts C1b, then because Alice has all the private keys used in the outputs of C1b, she can take the money. However, only Bob is able to broadcast C1b. To prevent this coin theft risk, Bob should

destroy all old Commitment Transactions.

Cooperatively Closing Out a Channel

Both parties are able to send as many payments to their counterparty as they wish, as long as they have funds available in the channel, knowing that in the event of disagreements they can broadcast to the blockchain the current state at any time.

In the vast majority of cases, all the outputs from the Funding Transaction will never be broadcast on the blockchain. They are just there in case the other party is non-cooperative, much like how a contract is rarely enforced in the courts. A proven ability for the contract to be enforced in a deterministic manner is sufficient incentive for both parties to act honestly.

When either party wishes to close out a channel cooperatively, they will be able to do so by contacting the other party and spending from the Funding Transaction with an output of the most current Commitment Transaction directly with no script encumbering conditions. No further payments may occur in the channel.

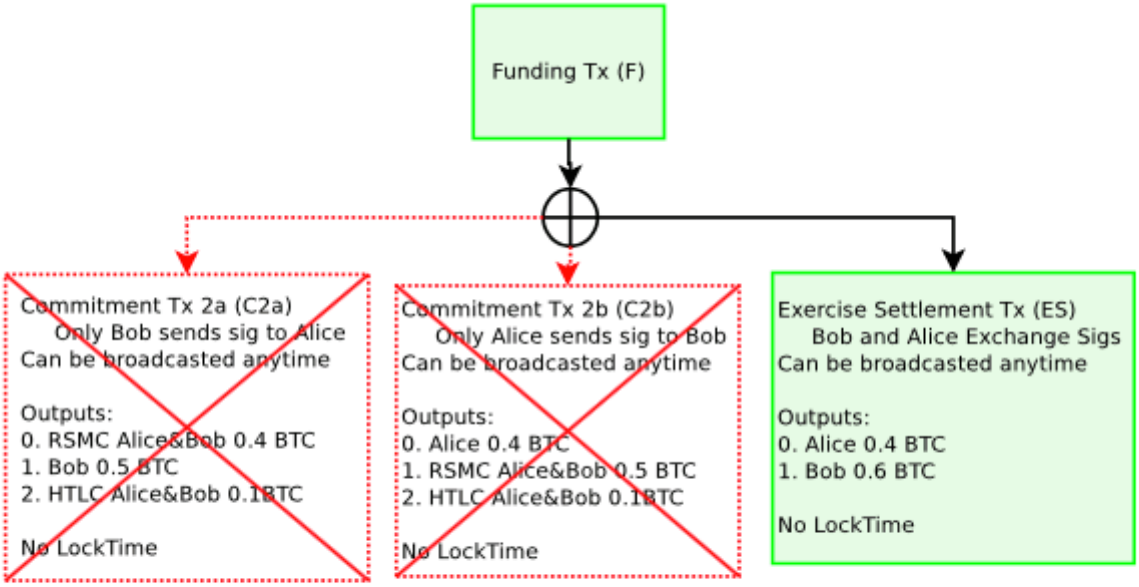


Figure 10: If both counterparties are cooperative, they take the balances in the current Commitment Transaction and spend from the Funding Transaction with a Exercise Settlement Transaction (ES). If the most recent Commitment Transaction gets broadcast instead, the payout (less fees) will be the same.

The purpose of closing out cooperatively is to reduce the number of transactions that occur on the blockchain and both parties will be able to receive their funds immediately (instead of one party waiting for the Revocation Delivery transaction to become valid).

Channels may remain in perpetuity until they decide to cooperatively close out the transaction, or when one party does not cooperate with another and the channel gets closed out and enforced on the blockchain.

Bidirectional Channel Implications and Summary

By ensuring channels can update only with the consent of both parties, it is possible to construct channels which perpetually exist in the blockchain. Both parties can update the balance inside the channel with whatever output balances they wish, so long as it's equal or less than the total funds committed inside the Funding Transaction; balances can move in both directions. If one party becomes malicious, either party may immediately close out the channel and broadcast the most current state to the blockchain. By using a fidelity

bond construction (Revocable Delivery Transactions), if a party violates the terms of the channel, the funds will be sent to the counterparty, provided the proof of violation (Breach Remedy Transaction) is entered into the blockchain in a timely manner. If both parties are cooperative, the channel can remain open indefinitely, possibly for many years.
This type of construction is only possible because adjudication occurs programatically over the blockchain as part of the Bitcoin consensus, so one does not need to trust the other party. As a result, one's channel counterparty does not possess full custody or control of the funds.

Hashed Timelock Contract (HTLC)

A bidirectional payment channel only permits secure transfer of funds inside a channel. To be able to construct secure transfers using a network of channels across multiple hops to the final destination requires an additional construction, a Hashed Timelock Contract (HTLC).
The purpose of an HTLC is to allow for global state across multiple nodes via hashes. This global state is ensured by time commitments and time-based unencumbering of resources via disclosure of preimages. Transactional "locking" occurs globally via commitments, at any point in time a single participant is responsible for disclosing to the next participant whether they have knowledge of the preimage R. This construction does not require custodial trust in one's channel counterparty, nor any other participant in the network.
In order to achieve this, an HTLC must be able to create certain transactions which are only valid after a certain date, using nLockTime, as well as information disclosure to one's channel counterparty. Additionally, this data must be revocable, as one must be able to undo an HTLC.
An HTLC is also a channel contract with one's counterparty which is enforcible via the blockchain. The counterparties in a channel agree to the following terms for a Hashed Timelock Contract:

1. If Bob can produce to Alice an unknown 20-byte random input data R from a known hash H, within three days, then Alice will settle the contract by paying Bob 0.1 BTC.
2. If three days have elapsed, then the above clause is null and void and the clearing process is invalidated, both parties must not attempt to settle and claim payment after three days.
3. Either party may (and should) pay out according to the terms of this contract in any method of the participants choosing and close out this contract early so long as both participants in this contract agree.
4. Violation of the above terms will incur a maximum penalty of the funds locked up in this contract, to be paid to the non-violating counterparty as a fidelity bond.

For clarity of examples, we use days for HTLCs and block height for RSMCs. In reality, the HTLC should also be defined as a block height (e.g. 3 days is equivalent to 432 blocks).
In effect, one desires to construct a payment which is contingent upon knowledge of R by the recipient within a certain timeframe. After this timeframe, the funds are refunded back to the sender.
Similar to RSMCs, these contract terms are programatically enforced on the Bitoin blockchain and do not require trust in the counterparty to adhere to the contract terms, as all violations are penalized via unilaterally enforced fidelity bonds, which are constructed using penalty transactions spending from commitment states. If Bob knows R within three days, then he can redeem the funds by broadcasting a transaction; Alice is unable to withhold the funds in any way, because the script returns as valid when the transaction is spent on the Bitcoin blockchain.
An HTLC is an additional output in a Commitment Transaction with a unique output script:

```
OP IF
OP ELSE
OP HASH160 <Hash160 (R)> OP EQUALVERIFY 2 <Alice2> <Bob2> OP CHECKMULTISIG
     2 <Alice1> <Bob1> OP CHECKMULTISIG OP ENDIF
```

Conceptually, this script has two possible paths spending from a single HTLC output. The first path (defined in the OP IF) sends funds to Bob if Bob can produce R. The second path is redeemed using a 3-day timelocked refund to Alice. The 3-day timelock is enforced using nLockTime from the spending transaction.

Non-revocable HTLC Construction

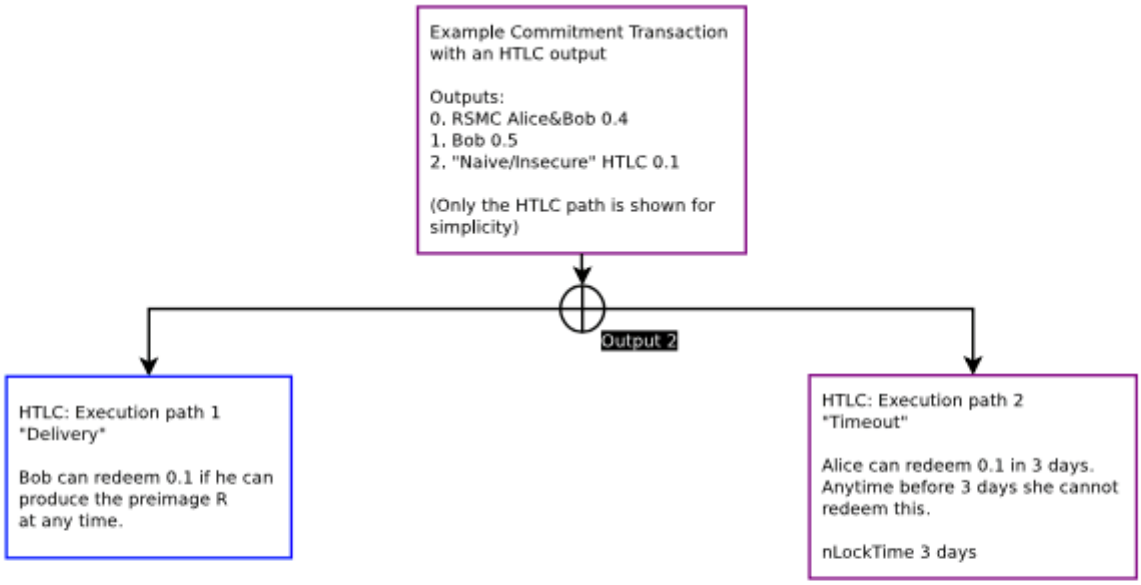


Figure 11: This is a non-functional naive implementation of an HTLC. Only the HTLC path from the Commitment Transaction is displayed. Note that there are two possible spends from an HTLC output. If Bob can produce the preimage R within 3 days and he can redeem path 1. After three days, Alice is able to broadcast path 2. When 3 days have elapsed either is valid. This model, however, doesn't work with multiple Commitment Transactions.

If R is produced within 3 days, then Bob can redeem the funds by broadcasting the "Delivery" transaction. A requirement for the "Delivery" transaction to be valid requires R to be included with the transaction. If R is not included, then the "Delivery" transaction is invalid. However, if 3 days have elapsed, the funds can be sent back to Alice by broadcasting transaction "Timeout". When 3 days have elapsed and R has been disclosed, either transaction may be valid.

It is within both parties individual responsibility to ensure that they can get their transaction into the blockchain in order to ensure the balances are correct. For Bob, in order to receive the funds, he must either broadcast the "Delivery" transaction on the Bitcoin blockchain, or otherwise settle with Alice (while cancelling the HTLC). For Alice, she must broadcast the "Timeout" 3 days from now to receive the refund, or cancel the HTLC entirely with Bob.

Yet this kind of simplistic construction has similar problems as an incorrect bidirectional payment channel construction. When an old Commitment Transaction gets broadcast, either party may attempt to steal funds as both paths may be valid after the fact. For example, if R gets disclosed 1 year later, and an incorrect

Commitment Transaction gets broadcast, both paths are valid and are redeemable by either party; the contract is not yet enforcible on the blockchain. Closing out the HTLC is absolutely necessary, because in order for Alice to get her refund, she must terminate the contract and receive her refund. Otherwise, when Bob discovers R after 3 days have elapsed, he may be able to steal the funds which should be going to Alice. With uncooperative counterparties it's not possible to terminate an HTLC without broadcasting it to the bitcoin blockchain as the uncooperative party is unwilling to create a new Commitment Transaction.

Off-chain Revocable HTLC

To be able to terminate this contract off-chain without a broadcast to the Bitcoin blockchain requires embedding RSMCs in the output, which will have a similar construction to the bidirectional channel.

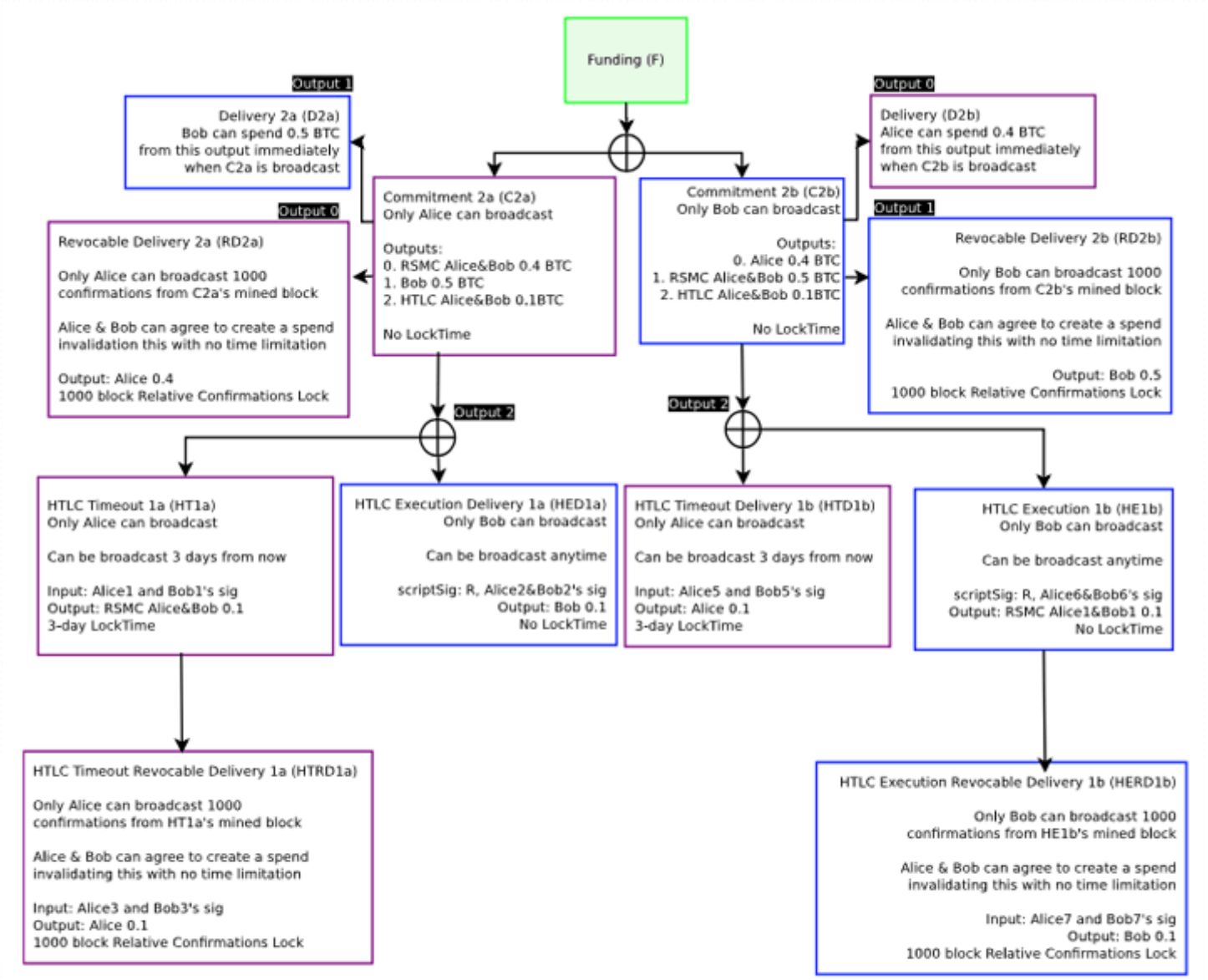


Figure 12: If Alice broadcasts C2a, then the left half will execute. If Bob broadcasts C2b, then the right half will execute. Either party may broadcast their Commitment transaction at any time. HTLC Timeout is only valid after 3 days. HTLC Executions can only be broadcast if the preimage to the hash R is known. Prior Commitments (and their dependent transactions) are not displayed for brevity.

Presume Alice and Bob wish to update their balance in the channel at Commitment 1 with a balance of 0.5 to

Alice and 0.5 to Bob.
Alice wishes to send 0.1 to Bob contingent upon knowledge of R within 3 days, after 3 days she wants her money back if Bob does not produce R.
The new Commitment Transaction will have a full refund of the current balance to Alice and Bob (Outputs 0 and 1), with output 2 being the HTLC, which describes the funds in transit. As 0.1 will be encumbered in an HTLC, Alice's balance is reduced to 0.4 and Bob's remains the same at 0.5.
This new Commitment Transaction (C2a/C2b) will have an HTLC output with two possible spends. Each spend is different depending on each counterparty's version of the Commitment Transaction. Similar to the bidirectional payment channel, when one party broadcasts their Commitment, payments to the counterparty will be assumed to be valid and not invalidated. This can occur because when one broadcasts a Commitment Transaction, one is attesting this is the most recent Commitment Transaction. If it is the most recent, then one is also attesting that the HTLC exists and was not invalidated before, so potential payments to one's counterparty should be valid.
Note that HTLC transaction names (beginning with the letter H) will begin with the number 1, whose values do not correlate with Commitment Transactions. This is simply the first HTLC transaction. HTLC transactions may persist between Commitment Transactions. Each HTLC has 4 keys per side of the transaction (C2a and C2b) for a total of 8 keys per counterparty.
The HTLC output in the Commitment Transaction has two sets of keys per counterparty in the output.
For Alice's Commitment Transaction (C2a), the HTLC output script requires multisig(PAlice2,PBob2) encumbered by disclosure of R, as well as multisig(PAlice1, PBob1) with no encumbering.
For Bob's Commitment Transaction (C2b), the HTLC output script requires multisig(PAlice6,PBob6) encumbered by disclosure of R, as well as multisig(PAlice5, PBob5) with no encumbering.
The HTLC output states are different depending upon which Commitment Transaction is broadcast.

HTLC when the Sender Broadcasts the Commitment Transaction

For the sender (Alice), the "Delivery" transaction is sent as an HTLC Execution Delivery transaction (HED1a), which is not encumbered in an RSMC. It assumes that this HTLC has never been terminated off-chain, as Alice is attesting that the broadcasted Commitment Transaction is the most recent. If Bob can produce the preimage R, he will be able to redeem funds from the HTLC after the Commitment Transaction is broadcast on the blockchain. This transaction consumes multisig(PAlice2,PBob2) if Alice broadcasts her Commitment C2a. Only Bob can broadcast HED1a since only Alice gave her signature for HED1a to Bob.
However, if 3 days have elapsed since forming the HTLC, then Alice will be able broadcast a "Timeout" transaction, the HTLC Timeout transaction (HT1a). This transaction is an RSMC. It consumes the output multisig(PAlice1,PBob1) without requiring disclosure of R if Alice broadcasts C2a. This transaction cannot enter into the blockchain until 3 days have elapsed. The output for this transaction is an RSMC with multisig(PAlice3,PBob3) with relative maturity of 1000 blocks, and multisig(PAlice4,PBob4) with no requirement for confirmation maturity. Only Alice can broadcast HT1a since only Bob gave his signature for HT1a to Alice.
After HT1a enters into the blockchain and 1000 block confirmations occur, an HTLC Timeout Revocable Delivery transaction (HTRD1a) may be broadcast by Alice which consumes multisig(PAlice3,PBob3). Only Alice can broadcast HTRD1a 1000 blocks after HT1a is broadcast since only Bob gave his signature for HTRD1a to Alice. This transaction can be revocable when another transaction supersedes HTRD1a using multisig(PAlice4,PBob4) which does not have any block maturity requirements.

HTLC when the Receiver Broadcasts the Commitment Transaction

For the potential receiver (Bob), the "Timeout" of receipt is refunded as an HTLC Timeout Delivery transaction (HTD1b). This transaction directly refunds the funds to the original sender (Alice) and is not encumbered in an RSMC. It assumes that this HTLC has never been terminated off-chain, as Bob is attesting that the broadcasted Commitment Transaction (C2b) is the most recent. If 3 days have elapsed, Alice can broadcast HTD1b and take the refund. This transaction consumes multisig(PAlice5,PAlice5) if Bob broadcasts C2b. Only Alice can broadcast HTD1b since Bob gave his signature for HTD1b to Alice.

However, if HTD1b is not broadcast (3 days have not elapsed) and Bob knows the preimage R, then Bob will be able to broadcast the HTLC Execution transaction (HE1b) if he can produce R. This transaction is an RSMC. It consumes the output multisig(PAlice6,PBob6) and requires disclosure of R if Bob broadcasts C2b. The output for this transaction is an RSMC with multisig(PAlice7,PBob7) with relative maturity of 1000 blocks, and multisig(PAlice8, PBob8) which does not have any block maturity requirements. Only Bob can broadcast HE1b since only Alice gave her signature for HE1b to Bob.

After HE1b enters into the blockchain and 1000 block confirmations occur, an HTLC Execution Revocable Delivery transaction (HERD1b) may be broadcast by Bob which consumes multisig(PAlice7,PBob7). Only Bob can broadcast HERD1b 1000 blocks after HE1b is broadcast since only Alice gave her signature for HERD1b to Bob. This transaction can be revocable when another transaction supersedes HERD1b using multisig(PAlice8,PBob8) which does not have any block maturity requirements.

HTLC Off-chain Termination

After an HTLC is constructed, to terminate an HTLC off-chain requires both parties to agree on the state of the channel. If the recipient can prove knowledge of R to the counterparty, the recipient is proving that they are able to immediately close out the channel on the Bitcoin blockchain and receive the funds. At this point, if both parties wish to keep the channel open, they should terminate the HTLC off-chain and create a new Commitment Transaction reflecting the new balance.

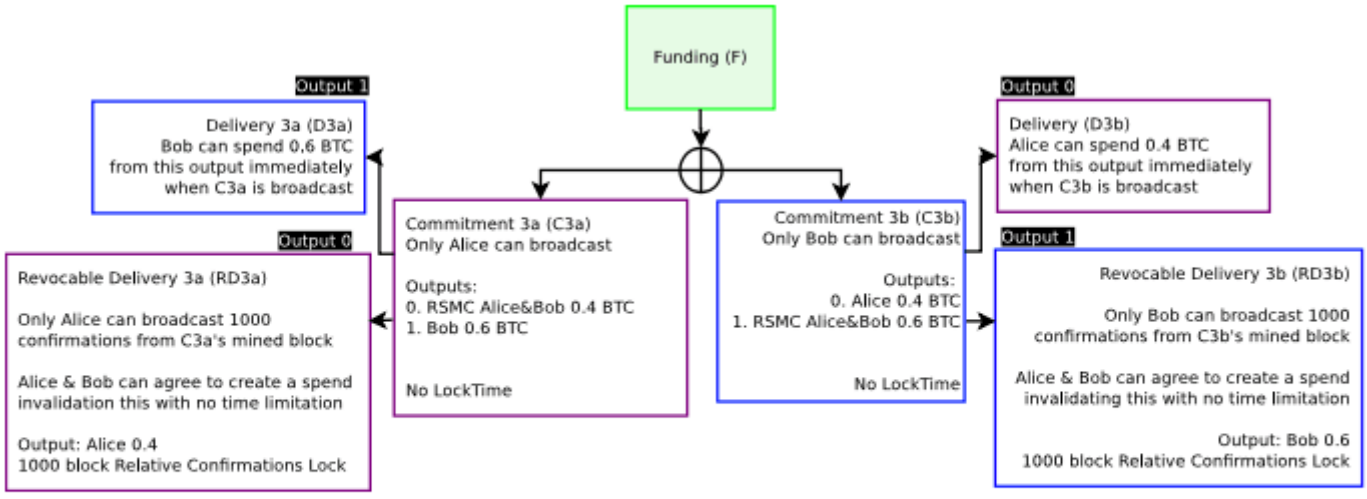


Figure 13: Since Bob proved to Alice he knows R by telling Alice R, Alice is willing to update the balance with a new Commitment Transaction. The payout will be the same whether C2 or C3 is broadcast at this time.

Similarly, if the recipient is not able to prove knowledge of R by disclosing R, both parties should agree to terminate the HTLC and create a new Commitment Transaction with the balance in the HTLC refunded to the sender.

If the counterparties cannot come to an agreement or become otherwise unresponsive, they should close out the channel by broadcasting the necessary channel transactions on the Bitcoin blockchain.

However, if they are cooperative, they can do so by first generating a new Commitment Transaction with the new balances, then invalidate the prior Commitment by exchanging Breach Remedy transactions (BR2a/BR2b). Additionally, if they are terminating a particular HTLC, they should also exchange some of their own private keys used in the HTLC transactions.

For example, Alice wishes to terminate the HTLC, Alice will disclose KAlice1 and KAlice4 to Bob. Correspondingly if Bob wishes to terminate the HTLC, Bob will disclose KBob6 and KBob8 to Alice. After the private keys are disclosed to the counterparty, if Alice broadcasts C2a, Bob will be able to take all the funds from the HTLC immediately. If Bob broadcasts C2b, Alice will be able to take all funds from the HTLC immediately. Note that when an HTLC is terminated, the older Commitment Transaction must be revoked as well.

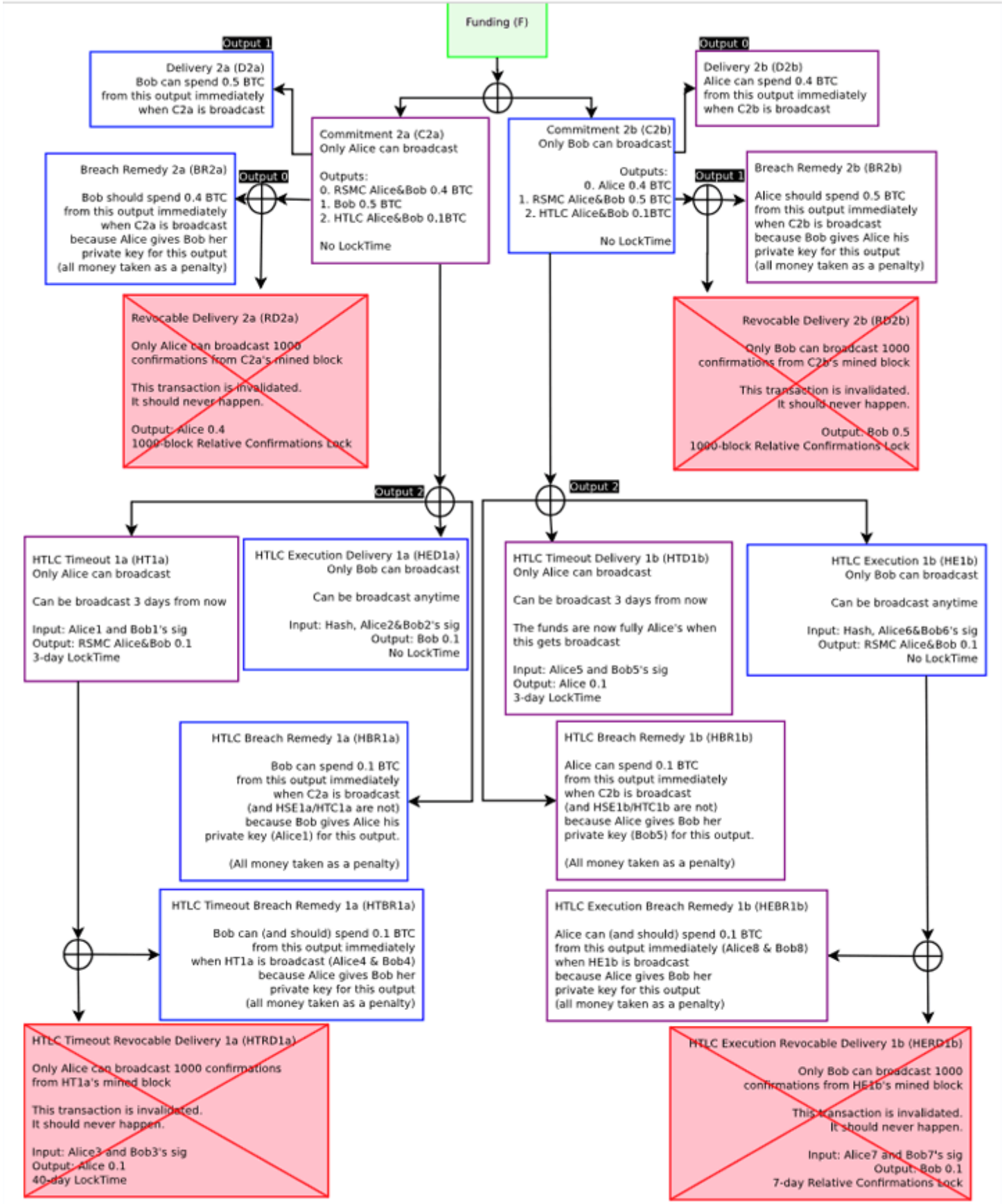


Figure 14: A fully revoked Commitment Transaction and terminated HTLC. If either party broadcasts Commitment 2, they will lose all their money to the counterparty. Other commitments (e.g. if Commitment 3 is the current Commitment) are not displayed for brevity.

Since both parties are able to prove the current state to each other, they can come to agreement on the current

balance inside the channel. Since they may broadcast the current state on the blockchain, they are able to come to agreement on netting out and terminating the HTLC with a new Commitment Transaction.

HTLC Formation and Closing Order

To create a new HTLC, it is the same process as creating a new Commitment Transaction, except the signatures for the HTLC are exchanged before the new Commitment Transaction's signatures.
To close out an HTLC, the process is as follows (from C2 to C3):

1. Alice signs and sends her signature for RD3b and C3b. At this point Bob can elect to broadcast C3b or C2b (with the HTLC) with the same payout. Bob is willing after receiving C3b to close out C2b.
2. Bob signs and sends his signature for RD3a and C3a, as well as his private keys used for Commitment 2 and the HTLC being terminated; he sends Alice KBobRSMC2, KBob5, and KBob8. At this point Bob should only broadcast C3b and should not broadcast C2b as he will lose all his money if he does so. Bob has fully revoked C2b and the HTLC. Alice is willing after receiving C3a to close out C2b.
3. Alice signs and sends her signature for RD3b and C3b, as well as her private keys used for Commitment 2 and the HTLC being terminated; she sends Bob KAliceRSMC2, KBob1, and KBob4. At this point neither party should broadcast Commitment 2, if they do so, their funds will be going to the counterparty. The old Commitment and old HTLC are now revoked and fully terminated. Only the new Commitment 3 remains, which does not have an HTLC.

When the HTLC has been closed, the funds are updated so that the present balance in the channel is what would occur had the HTLC contract been completed and broadcast on the blockchain. Instead, both parties elect to do off-chain novation and update their payments inside the channel.
It is absolutely necessary for both parties to complete off-chain novation within their designated time window. For the receiver (Bob), he must know R and update his balance with Alice within 3 days (or whatever time was selected), else Alice will be able to redeem it within 3 days. For Alice, very soon after her timeout becomes valid, she must novate or broadcast the HTLC Timeout transaction. She must also novate or broadcast the HTLC Timeout Revocable Delivery transaction as soon as it becomes valid. If the counterparty is unwilling to novate or is stalling, then one must broadcast the current channel state, including HTLC transactions) onto the Bitcoin blockchain.
The amount of time flexibility with these offers to novate are dependent upon one's contingent dependencies on the hashlock R. If one establishes a contract that the HTLC must be resolved within 1 day, then if the transaction times out Alice must resolve it by day 4 (3 days plus 1), else Alice risks losing funds.

Key Storage

Keys are generated using BIP 0032 Hierarchical Deterministic Wallets. Keys are pre-generated by both parties. Keys are generated in a merkle tree and are very deep within the tree. For instance, Alice pre-generates one million keys, each key being a child of the previous key. Alice allocates which keys to use according to some deterministic manner. For example, she starts with the child deepest in the tree to generate many sub-keys for day 1. This key is used as a master key for all keys generated on day 1. She gives Bob the address she wishes to use for the next transaction, and discloses the private key to Bob when it becomes invalidated. When Alice discloses to Bob all private keys derived from the day 1 master key and does not wish to continue using that master key, she can disclose the day 1 master key to Bob. At this point, Bob does not need to store all the keys derived from the day 1 master key. Bob does the same for Alice and gives her his day 1 key.
When all Day 2 private keys have been exchanged, for example by day 5, Alice discloses her Day 2 key. Bob

is able to generate the Day 1 key from the Day 2 key, as the Day 1 key is a child of the Day 2 key as well.
If a counterparty broadcasts the wrong Commitment Transaction, which private key to use in a transaction to recover funds can either be brute forced, or if both parties agree, they can use the sequence id number when creating the transaction to identify which sets of keys are used.
This enables participants in a channel to have prior output states (transactions) invalidated by both parties without using much data at all. By disclosing private keys pre-arranged in a merkle-tree, it is possible to invalidate millions of old transactions with only a few kilobytes of data per channel. Core channels in the KRABLR Network can conduct billions of transactions without a need for significant storage costs.

Blockchain Transaction Fees for Bidirectional Channels

It is possible for each participant to generate different versions of transactions to ascribe blame as to who broadcast the transaction on the blockchain. By having knowledge of who broadcast a transaction and the ability to ascribe blame, a third party service can be used to hold fees in a 2-of-3 multisig escrow. If one wishes to broadcast the transaction chain instead of agreeing to do a Funding Close or replacement with a new Commitment Transaction, one would communicate with the third party and broadcast the chain to the blockchain. If the counterparty refuses the notice from the third party to cooperate, the penalty is rewarded to the non-cooperative party. In most instances, participants may be indifferent to the transaction fees in the event of an uncooperative counterparty.
One should pick counterparties in the channel who will be cooperative, but is not an absolute necessity for the system to function. Note that this does not require trust among the rest of the network, and is only relevant for the comparatively minor transaction fees. The less trusted party may just be the one responsible for transaction fees.
The KRABLR Network fees will likely be significantly lower than blockchain transaction fees. The fees are largely derived from the time-value of locking up funds for a particular route, as well as paying for the chance of channel close on the blockchain. These should be significantly lower than on-chain transactions, as many transactions on a KRABLR Network channel can be settled into one single blockchain transaction. With a sufficiently robust and interconnected network, the fees should asymptotically approach negligibility for many types of transactions. With cheap fees and fast transactions, it will be possible to build scalable micropayments, even amongst high-frequency systems such as Internet of Things applications or per-unit micro-billing.

Pay to Contract

It is possible construct a cryptographically provable "Delivery Versus Payment" contract, or pay-to-contract, as proof of payment. This proof can be established as knowledge of the input R from hash(R) as payment of a certain value. By embedding a clause into the contract between the buyer and seller stating that knowing R is proof of funds sent, the recipient of funds has no incentive to disclose R unless they have certainty that they will receive payment. When the funds eventually get pulled from the buyer by their counterparty in their micropayment channel, R is disclosed as part of that pull of funds. One can design paper legal documents that specify that knowledge or disclosure of R implies fulfillment of payment. The sender can then arrange a cryptographically signed contract with knowledge of inputs for hashes treated as fulfillment of the paper contract before payment occurs.

The KRABLR Network

By having a micropayment channel with contracts encumbered by hashlocks and timelocks, it is possible to

clear transactions over a multi-hop payment network using a series of decrementing timelocks without additional central clearinghouses.

Traditionally, financial markets clear transactions by transferring the obligation for delivery at a central point and settle by transferring ownership through this central hub. Bank wire and fund transfer systems (such as ACH and the Visa card network), or equities clearinghouses (such as the DTCC) operate in this manner.

As Bitcoin enables programmatic money, it is possible to create transactions without contacting a central clearinghouse. Transactions can execute off-chain with no third party which collects all funds before disbursing it â€“ only transactions with uncooperative channel counterparties become automatically adjudicated on the blockchain.

The obligation to deliver funds to an end-recipient is achieved through a process of chained delegation. Each participant along the path assumes the obligation to deliver to a particular recipient. Each participant passes on this obligation to the next participant in the path. The obligation of each subsequent participant along the path, defined in their respective HTLCs, has a shorter time to completion compared to the prior participant. This way each participant is sure that they will be able to claim funds when the obligation is sent along the path.

Bitcoin Transaction Scripting, a form of what some call an implementation of "Smart Contracts", enables systems without trusted custodial clearinghouses or escrow services.

Decrementing Timelocks

Presume Alice wishes to send 0.001 BTC to Dave. She locates a route through Bob and Carol. The transfer path would be Alice to Bob to Carol to Dave.

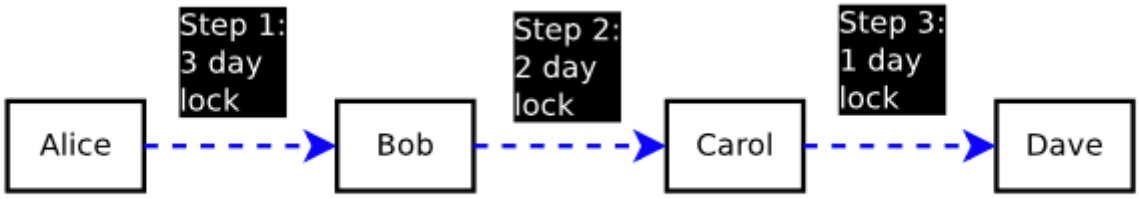


Figure 15: Payment over the Lightning Network using HTLCs.

When Alice sends payment to Dave through Bob and Carol, she requests from Dave hash(R) to use for this payment. Alice then counts the amount of hops until the recipient and uses that as the HTLC expiry. In this case, she sets the HTLC expiry at 3 days. Bob then creates an HTLC with Carol with an expiry of 2 days, and Carol does the same with Dave with an expiry of 1 day. Dave is now free to disclose R to Carol, and both parties will likely agree to immediate settlement via novation with a replacement Commitment Transaction. This then occurs step-by-step back to Alice. Note that this occurs off-chain, and nothing is broadcast to the blockchain when all parties are cooperative.

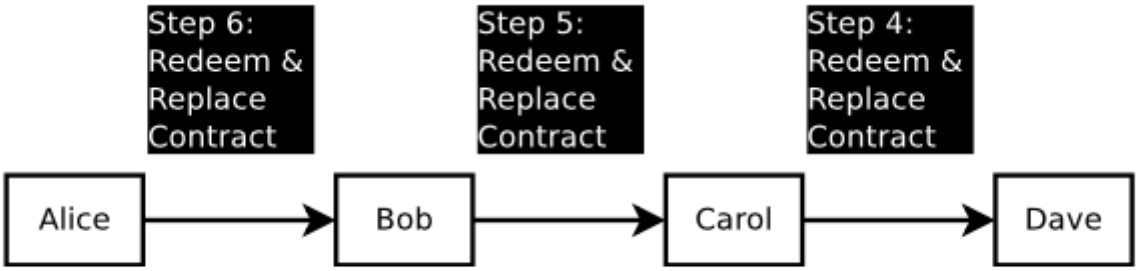


Figure 16: Settlement of HTLC, Alice's funds get sent to Dave.

Decrementing timelocks are used so that all parties along the path know that the disclosure of R will allow the disclosing party to pull funds, since they will at worst be pulling funds after the date whereby they must receive R. If Dave does not produce R within 1 day to Carol, then Carol will be able to close out the HTLC. If Dave broadcasts R after 1 day, then he will not be able to pull funds from Carol. Carol's responsibility to Bob occurs on day 2, so Carol will never be responsible for payment to Dave without an ability to pull funds from Bob provided that she updates her transaction with Dave via transmission to the blockchain or via novation.
In the event that R gets disclosed to the participants halfway through expiry along the path (e.g. day 2), then it is possible for some parties along the path to be enriched. The sender will be able to know R, so due to Pay to Contract, the payment will have been fulfilled even though the receiver did not receive the funds. Therefore, the receiver must never disclose R unless they have received an HTLC from their channel counterparty; they are guaranteed to receive payment from one of their channel counterparties upon disclosure of the preimage.
In the event a party outright disconnects, the counterparty will be responsible for broadcasting the current Commitment Transaction state in the channel to the blockchain. Only the failed non-responsive channel state gets closed out on the blockchain, all other channels should continue to update their Commitment Transactions via novation inside the channel. Therefore, counterparty risk for transaction fees are only exposed to direct channel counterparties. If a node along the path decides to become unresponsive, the participants not directly connected to that node suffer only decreased time-value of their funds by not conducting early settlement before the HTLC close.

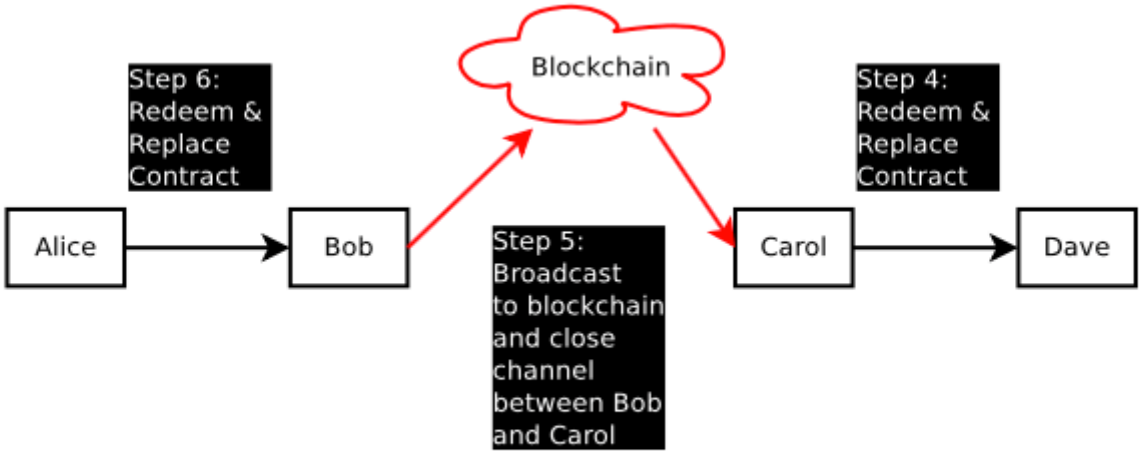


Figure 17: Only the non-responsive channels get broadcast on the blockchain, all others are settled off-chain via novation.

Payment Amount

It is preferable to use a small payment per HTLC. One should not use an extremely high payment, in case the payment does not fully route to its destination. If the payment does not reach its destination and one of the participants along the path is uncooperative, it is possible that the sender must wait until the expiry before receiving a refund. Delivery may be lossy, similar to packets on the internet, but the network cannot outright steal funds in transit. Since transactions don't hit the blockchain with cooperative channel counterparties, it is recommended to use as small of a payment as possible. A tradeoff exists between locking up transaction fees on each hop versus the desire to use as small a transaction amount as possible (the latter of which may incur higher total fees). Smaller transfers with more intermediaries imply a higher percentage paid as KRABLR Network fees to the intermediaries.

Clearing Failure and Rerouting

If a transaction fails to reach its final destination, the receiver should send an equal payment to the sender with the same hash, but not disclose R. This will net out the disclosure of the hash for the sender, but may not for the receiver. The receiver, who generated the hash, should discard R and never broadcast it. If one channel along the path cannot be contacted, then the channels may elect to wait until the path expires, which all participants will likely close out the HTLC as unsettled without any payment with a new Commitment Transaction.

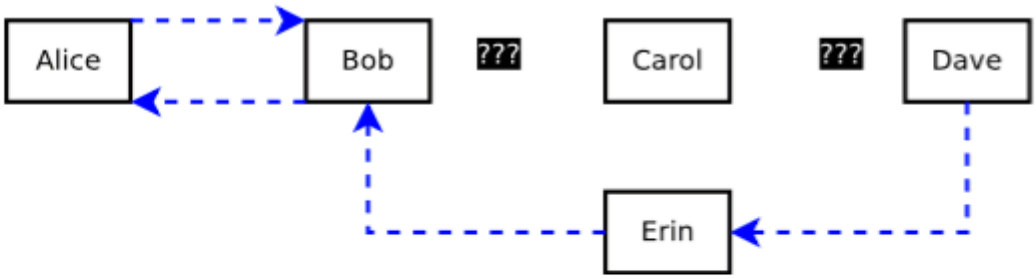


Figure 18: Dave creates a path back to Alice after Alice fails to send funds to Dave, because Carol is uncooperative. The input R from hash(R) is never brodcast by Dave, because Carol did not complete her actions. If R was broadcast, Alice will break-even. Dave, who controls R should never broadcast R because he may not receive funds from Carol, he should let the contracts expire. Alice and Bob have the option to net out and close the contract early, as well, in this diagram.

If the refund route is the same as the payment route, and there are no half-signed contracts whereby one party may be able to steal funds, it is possible to outright cancel the transaction by replacing it with a new Commitment Transaction starting with the most recent node who participated in the HTLC.

It is also possible to clear out a channel by creating an alternate route path in which payment will occur in the opposite direction (netting out to zero) and/or creating an entirely alternate route for the payment path. This will create a time-value of money for disclosing inputs to hashes on the KRABLR Network. Participants may specialize in high connectivity between nodes and offering to offload contract hashlocks from other nodes for a fee. These participants will agree to payments which net out to zero (plus fees), but are loaning bitcoins for a set time period. Most likely, these entities with low demand for channel resources will be end-users who are already connected to multiple well-connected nodes. When an end-user connects to a node, the node may ask the client to lock up their funds for several days to another channel the client has established for a fee. This can be achieved by having the new transactions require a new hash(Y) from input Y in addition to the existing hash which may be generated by any participant, but must disclose Y only after a full circle is established. The new participant has the same responsibility as well as the same timelocks as the old participant being replaced. It is also possible that the one new participant replaces multiple hops.

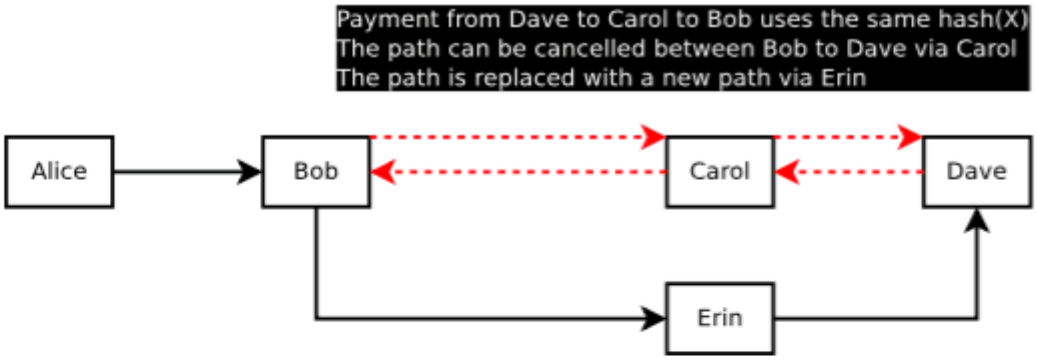


Figure 19: Erin is connected to both Bob and Dave. If Bob wishes to free up his channel with Carol, since that channel is active and very profitable, Bob can offload the payment to Dave via Erin. Since Erin has extra bitcoin available, she will be able to collect some fee for offloading the channel between Bob and Carol as well as between Carol and Dave. The channels between Bob and Carol as well as Carol and Dave are undone and no longer have the HTLC, nor has payment occurred on that path. Payment will occur on the path involving Erin. This is achieved by creating a new payment from Dave to Carol to Bob contingent upon Erin constructing an HTLC. The payment in dashed lines (red) are netted out to zero and settled via a new Commitment Contract.

Payment Routing

It is theoretically possible to build a route map implicitly from observing 2-of-2 multisigs on the blockchain to build a routing table. Note, however, this is not feasible with pay-to-script-hash transaction outputs, which can be resolved out-of-band from the bitcoin protocol via a third party routing service. Building a routing table will become necessary for large operators (e.g. BGP, Cjdns). Eventually, with optimizations, the network will look a lot like the correspondent banking network, or Tier-1 ISPs. Similar to how packets still reach their destination on your home network connection, not all participants need to have a full routing table. The core Tier-1 routes can be online all the time -while nodes at the edges, such as average users, would be connected intermittently.

Node discovery can occur along the edges by pre-selecting and offering partial routes to well-known nodes.

Risks

The primary risks relate to timelock expiration. Additionally, for core nodes and possibly some merchants to be able to route funds, the keys must be held online for lower latency. However, end-users and nodes are able to keep their private keys firewalled off in cold storage.

Improper Timelocks

Participants must choose timelocks with sufficient amounts of time. If insufficient time is given, it is possible that timelocked transactions believed to be invalid will become valid, enabling coin theft by the counterparty. There is a trade-off between longer timelocks and the time-value of money. When writing wallet and KRABLR Network application software, it is necessary to ensure that sufficient time is given and users are able to have their transactions enter into the blockchain when interacting with non-cooperative or malicious

channel counterparties.

Forced Expiration Spam

Forced expiration of many transactions may be the greatest systemic risk when using the KRABLR Network. If a malicious participant creates many channels and forces them all to expire at once, these may overwhelm block data capacity, forcing expiration and broadcast to the blockchain. The result would be mass spam on the bitcoin network. The spam may delay transactions to the point where other locktimed transactions become valid.

This may be mitigated by permitting one transaction replacement on all pending transactions. Anti-spam can be used by permitting only one transaction replacement of a higher sequence number by the inverse of an even or odd number. For example, if an odd sequence number was broadcast, permit a replacement to a higher even number only once. Transactions would use the sequence number in an orderly way to replace other transactions. This mitigates the risk assuming honest miners. This attack is extremely high risk, as incorrect broadcast of Commitment Transactions entail a full penalty of all funds in the channel.

Additionally, one may attempt to steal HTLC transactions by forcing a timeout transaction to go through when it should not. This can be easily mitigated by having each transfer inside the channel be lower than the total transaction fees used. Since transactions are extremely cheap and do not hit the blockchain with cooperative channel counterparties, large transfers of value can be split into many small transfers. This attempt can only work if the blocks are completely full for a long time. While it is possible to mitigate it using a longer HTLC timeout duration, variable block sizes may become common, which may need mitigations.

If this type of transaction becomes the dominant form of transactions which are included on the blockchain, it may become necessary to increase the block size and run a variable blocksize structure and timestop flags as described in the section below. This can create sufficient penalties and disincentives to be highly unprofitable and unsuccessful for attackers, as attackers lose all their funds from broadcasting the wrong transaction, to the point where it will never occur.

Coin Theft via Cracking

As parties must be online and using private keys to sign, there is a possibility that, if the computer where the private keys are stored is compromised, coins will be stolen by the attacker. While there may be methods to mitigate the threat for the sender and the receiver, the intermediary nodes must be online and will likely be processing the transaction automatically. For this reason, the intermediary nodes will be at risk and should not be holding a substantial amount of money in this "hot wallet." Intermediary nodes which have better security will likely be able to out-compete others in the long run and be able to conduct greater transaction volume due to lower fees. Historically, one of the largest component of fees and interest in the financial system are from various forms of counterparty risk â€“ in Bitcoin it is possible that the largest component in fees will be derived from security risk premiums.

A Funding Transaction may have multiple outputs with multiple Commitment Transactions, with the Funding Transaction key and some Commitment Transactions keys stored offline. It is possible to create an equivalent of a "Checking Account" and "Savings Account" by moving funds between outputs from a Funding Transaction, with the "Savings Account" stored offline and requiring additional signatures from security services.

Data Loss

When one party loses data, it is possible for the counterparty to steal funds. This can be mitigated by having a

third party data storage service where encrypted data gets sent to this third party service which the party cannot decrypt. Additionally, one should choose channel counterparties who are responsible and willing to provide the current state, with some periodic tests of honesty.

Forgetting to Broadcast the Transaction in Time

If one does not broadcast a transaction at the correct time, the counterparty may steal funds. This can be mitigated by having a designated third party to send funds. An output fee can be added to create an incentive for this third party to watch the network. Further, this can also be mitigated by implementing OP CHECKSEQUENCEVERIFY.

Inability to Make Necessary Soft-Forks

Changes are necessary to bitcoin, such as the malleability soft-fork. Additionally, if this system becomes popular, it will be necessary for the system to securely transact with many users and some kind of structure like a blockheight timestop will be desirable. This system assumes such changes to enable KRABLR Network to exist entirely, as well as soft-forks ensuring the security is robust against attackers will occur. While the system may continue to operate with only some time lock and malleability soft-forks, there will be necessary soft-forks regarding systemic risks. Without proper community foresight, an inability to establish a timestop or similar function will allow systemic attacks to take place and may not be recognized as imperative until an attack actually occurs.

Colluding Miner Attacks

Miners may elect to refuse to enter in particular transactions (e.g. Breach Remedy transactions) in order to assist in timeout coin theft. An attacker can pay off all miners to refuse to include certain transactions in their mempool and blocks. The miners can identify their own blocks in an attempt to prove their behavior to the paying attacker.

This can be mitigated by encouraging miners to avoid identifying their own blocks. Further, it should be expected that this kind of payment to miners is malicious activity and the contract is unenforcible. Miners may then take payment and surreptitiously mine a block without identifying the block to the attacker. Since the attacker is paying for this, they will quickly run out of money by losing the fee to the miner, as well as losing all their money in the channel. This attack is unlikely and fairly unattractive as it is far too difficult and requires a high degree of collusion with extreme risk.

The risk model of this attack occurirng is similar to that of miners colluding to do reorg attacks: Extremely unlikely with many uncoordinated miners.

Block Size Increases and Consensus

If we presume that a decentralized payment network exists and one user will make 3 blockchain transactions per year on average, Bitcoin will be able to support over 35 million users with 1MB blocks in ideal circumstances (assuming 2000 transactions/MB, or 500 bytes/Tx). This is quite limited, and an increase of the block size may be necessary to support everyone in the world using Bitcoin. A simple increase of the block size would be a hard fork, meaning all nodes will need to update their wallets if they wish to participate in the network with the larger blocks.

While it may appear as though this system will mitigate the block size increases in the short term, if it achieves global scale, it will necessitate a block size increase in the long term. Creating a credible tool to help prevent blockchain spam designed to encourage transactions to timeout becomes imperative.

To mitigate timelock spam vulnerabilities, non-miner and miners' consensus rules may also differ if the miners' consensus rules are more restrictive. Non-miners may accept blocks over 1MB, while miners may have different soft-caps on block sizes. If a block size is above that cap, then that is viewed as an invalid block by other miners, but not by non-miners. The miners will only build the chain on blocks which are valid according to the agreed-upon soft-cap. This permits miners to agree on raising the block size limit without requiring frequent hard-forks from clients, so long as the amount raised by miners does not go over the clients' hard limit. This mitigates the risk of mass expiry of transactions at once. All transactions which are not redeemed via Exercise Settlement (ES) may have a very high fee attached, and miners may use a consensus rule whereby those transactions are exempted from the soft-cap, making it very likely the correct transactions will enter the blockchain.

When transactions are viewed as circuits and contracts instead of transaction packets, the consensus risks can be measured by the amount of time available to cover the UTXO set controlled by hostile parties. In effect, the upper bound of the UTXO size is determined by transaction fees and the standard minimum transaction output value. If the bitcoin miners have a deterministic mempool which prioritizes transactions respecting a "weak" local time order of transactions, it could become extremely unprofitable and unlikely for an attack to succeed. Any transaction spam time attack by broadcasting the incorrect Commitment Transaction is extremely high risk for the attacker, as it requires an immense amount of bitcoin and all funds committed in those transactions will be lost if the attacker fails.

Use Cases

In addition to helping bitcoin scale, there are many uses for transactions on the KRABLR Network:

```
Instant Transactions. Using KRABLR, Bitcoin transactions are now nearly
instant with any party. It is possible to pay for a cup of coffee with
direct non-revocable payment in milliseconds to seconds.
```

```
Exchange Arbitrage. There is presently incentive to hold funds on
exchanges to be ready for large market moves due to 3-6 block confirmation
times. It is possible for the exchange to participate in this network and
for clients to move their funds on and off the exchange for orders nearly
instantly. If the exchange does not have deep market depth and commits to
only permitting limit orders close to the top of the order book, then the
risk of coin theft becomes much lower. The exchange, in effect, would no
longer have any need for a cold storage wallet. This may substantially
reduce thefts and the need for trusted third party custodians.
```

```
Micropayments. Bitcoin blockchain fees are far too high to accept
micropayments, especially with the smallest of values. With this system,
near-instant micropayments using Bitcoin without a 3rd party custodian
would be possible. It would enable, for example, paying per-megabyte for
internet service or per-article to read a newspaper.
```

Financial Smart Contracts and Escrow. Financial contracts are especially time-sensitive and have higher demands on blockchain computation. By moving the overwhelming majority of trustless transactions off-chain, it is possible to have highly complex transaction contract terms without ever hitting the blockchain.

Cross-Chain Payments. So long as there are similar hash-functions across chains, it's possible for transactions to be routed over multiple chains with different consensus rules. The sender does not have to trust or even know about the other chains â€“ even the destination chain. Simiarly, the receiver does not have to know anything about the sender's chain or any other chain. All the receiver cares about is a conditional payment upon knowledge of a secret on their chain. Payment can be routed by participants in both chains in the hop. E.g. Alice is on Bitcoin, Bob is on both Bitcoin and X-Coin and Carol is on a hypothetical X-Coin, Alice can pay Carol without understanding the X-Coin consensus rules.

Calculations

We consider the scenario of an attacker trying to generate an alternate chain faster than the honest chain. Even if this is accomplished, it does not throw the system open to arbitrary changes, such as creating value out of thin air or taking money that never belonged to the attacker. Nodes are not going to accept an invalid transaction as payment, and honest nodes will never accept a block containing them. An attacker can only try to change one of his own transactions to take back money he recently spent.

The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1.

The probability of an attacker catching up from a given deficit is analogous to a Gambler's Ruin problem. Suppose a gambler with unlimited credit starts at a deficit and plays potentially an infinite number of trials to try to reach breakeven. We can calculate the probability he ever reaches breakeven, or that an attacker ever catches up with the honest chain, as follows:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \leq q \\ (q/p)^z & \text{if } p > q \end{cases}$$

Given our assumption that p > q, the probability drops exponentially as the number of blocks the attacker has to catch up with increases. With the odds against him, if he doesn't make a lucky lunge forward early on, his chances become vanishingly small as he falls further behind.

We now consider how long the recipient of a new transaction needs to wait before being sufficiently certain

the sender can't change the transaction. We assume the sender is an attacker who wants to make the recipient believe he paid him for a while, then switch it to pay back to himself after some time has passed. The receiver will be alerted when that happens, but the sender hopes it will be too late.

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z\frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Converting to C code...

```
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Running some results, we can see the probability drop off exponentially with z.

```
q=0.1
z=0    P=1.0000000
z=1    P=0.2045873
z=2    P=0.0509779
z=3    P=0.0131722
z=4    P=0.0034552
z=5    P=0.0009137
z=6    P=0.0002428
z=7    P=0.0000647
z=8    P=0.0000173
z=9    P=0.0000046
z=10   P=0.0000012

q=0.3
z=0    P=1.0000000
z=5    P=0.1773523
z=10   P=0.0416605
z=15   P=0.0101008
z=20   P=0.0024804
z=25   P=0.0006132
z=30   P=0.0001522
z=35   P=0.0000379
z=40   P=0.0000095
z=45   P=0.0000024
z=50   P=0.0000006
```

Solving for P less than 0.1%...

```
P < 0.001
q=0.10   z=5
q=0.15   z=8
q=0.20   z=11
q=0.25   z=15
q=0.30   z=24
q=0.35   z=41
q=0.40   z=89
q=0.45   z=340
```

Questions and Intuition

Here are some questions that since these weeks, dreams asked me and I woke up sweating. But in fact it is OK.

```
Q. If you delete the transaction outputs, user cannot verify the
rangeproof and maybe a negative amount is created.

A. This is OK. For the entire transaction to validate all negative amounts
must have been destroyed. User have SPV security only that no illegal
inflation happened in the past, but the user knows that _at this time_ no
inflation occurred.
```

```
Q. If you delete the inputs, double spending can happen.
```

```
A. In fact, this means: maybe someone claims that some unspent output was
spent in the old days. But this is impossible, otherwise the sum of the
combined transaction could not be zero.

An exception is that if the outputs are amount zero, it is possible to
make two that are negatives of each other, and the pair can be revived
without anything breaks. So to prevent consensus problems, outputs
0-amount should be banned. Just add H at each output, now they all amount
to at least 1.
```

Future Research

Here are some questions I can not answer at the time of this writing.

```
1. What script support is possible? We would need to translate script
operations into some sort of discrete logarithm information.
```

```
2. We require user to check all k*G values, when in fact all that is
needed is that their sum is of the form k*G. Instead of using signatures
is there another proof of discrete logarithm that could be combined?
```

```
3. There is a denial-of-service option when a user downloads the chain,
the peer can give gigabytes of data and list the wrong unspent outputs.
The user will see that the result do not add up to 0, but cannot tell
where the problem is.

For now maybe the user should just download the blockchain from a Torrent
or something where the data is shared between many users and is reasonably
likely to be correct.
```

Conclusion

We have proposed a system for electronic transactions without relying on trust. We started with the usual framework of coins made from digital signatures, which provides strong control of ownership, but is incomplete without a way to prevent double-spending. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The network is robust in its unstructured simplicity. Nodes work all at once with little coordination. They do not need to be identified, since messages are not routed to any particular place and only need to be delivered on a best effort basis. Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone. They vote with their CPU power, expressing their acceptance of valid blocks

by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism.

The KRABLR protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits, financial contracts, gambling markets and the like via a highly generalized programming language. The KRABLR protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. What is more interesting about KRABLR, however, is that the KRABLR protocol moves far beyond just currency. Protocols around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. Finally, there is also a substantial array of applications that have nothing to do with money at all.

The concept of an arbitrary state transition function as implemented by the KRABLR protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, KRABLR is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

Creating a network of micropayment channels enables bitcoin scalability, micropayments down to the satoshi, and near-instant transactions. These channels represent real Bitcoin transactions, using the Bitcoin scripting opcodes to enable the transfer of funds without risk of counterparty theft, especially with long-term miner risk mitigations.

If all transactions using Bitcoin were on the blockchain, to enable 7 billion people to make two transactions per day, it would require 24GB blocks every ten minutes at best (presuming 250 bytes per transaction and 144 blocks per day). Conducting all global payment transactions on the blockchain today implies miners will need to do an incredible amount of computation, severely limiting bitcoin scalability and full nodes to a few centralized processors.

If all transactions using Bitcoin were conducted inside a network of micropayment channels, to enable 7 billion people to make two channels per year with unlimited transactions inside the channel, it would require 133 MB blocks (presuming 500 bytes per transaction and 52560 blocks per year). Current generation desktop computers will be able to run a full node with old blocks pruned out on 2TB of storage.

With a network of instantly confirmed micropayment channels whose payments are encumbered by timelocks and hashlock outputs, Bitcoin can scale to billions of users without custodial risk or blockchain centralization when transactions are conducted securely off-chain using bitcoin scripting, with enforcement of non-cooperation by broadcasting signed multisignature transactions on the blockchain.

References

W. Dai, "b-money," http://www.weidai.com/bmoney.txt, 1998.

H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," In 20th Symposium on Information Theory in the Benelux, May 1999.

S. Haber, W.S. Stornetta, "How to time-stamp a digital document," In Journal of Cryptology, vol 3, no 2, pages 99-111, 1991.

D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," In Sequences II: Methods in Communication, Security and Computer Science, pages 329-334, 1993.

S. Haber, W.S. Stornetta, "Secure names for bit-strings," In Proceedings of the 4th ACM Conference on Computer and Communications Security, pages 28-35, April 1997.

A. Back, "Hashcash - a denial of service counter-measure," http://www.hashcash.org/papers/hashcash.pdf, 2002.

R.C. Merkle, "Protocols for public key cryptosystems," In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society, pages 122-133, April 1980.

W. Feller, "An introduction to probability theory and its applications," 1957.

Koinster, "White Paper Generator," http://whitepaper.koinster.com, 2017.

Intrinsic value: http://bitcoinmagazine.com/8640/an-exploration-of-intrinsic-value-what-it-is-why-bitcoin-doesnt-have-it-and-why-bitcoin-does-have-it/

Smart property: https://en.bitcoin.it/wiki/Smart_Property

Smart contracts: https://en.bitcoin.it/wiki/Contracts

B-money: http://www.weidai.com/bmoney.txt

Reusable proofs of work: http://www.finney.org/~hal/rpow/

Secure property titles with owner authority: http://szabo.best.vwh.net/securetitle.html

Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf

Namecoin: https://namecoin.org/

Zooko's triangle: http://en.wikipedia.org/wiki/Zooko's_triangle

Colored coins whitepaper: https://docs.google.com/a/buterin.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit

Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec

Decentralized autonomous corporations, Bitcoin Magazine: http://bitcoinmagazine.com/7050/bootstrapping-a-decentralized-autonomous-corporation-part-i/

Simplified payment verification: https://en.bitcoin.it/wiki/Scalability#Simplifiedpaymentverification

Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree

Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree

GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf

StorJ and Autonomous Agents, Jeff Garzik:
http://garzikrants.blogspot.ca/2013/01/storj-and-bitcoin-autonomous-agents.html

Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y

Ethereum RLP: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-RLP

Ethereum Merkle Patricia trees: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-Patricia-Tree

Peter Todd on Merkle sum trees: http://sourceforge.net/p/bitcoin/mailman/message/31709140

https://people.xiph.org/~greg/confidential_values.txt

https://bitcointalk.org/index.php?topic=279249.0

https://cryptonote.org/whitepaper.pdf

https://eprint.iacr.org/2015/1098.pdf

https://download.wpsoftware.net/bitcoin/wizardry/horasyuanmouton-owas.pdf

http://blockstream.com/sidechains.pdf

http://fr.harrypotter.wikia.com/wiki/Sortilege_de_Langue_de_Plomb

https://bitcointalk.org/index.php?topic=281848.0

Satoshi Nakamoto. Bitcoin: A Peer-to-peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, Oct 2008.

Manny Trillo. Stress Test Prepares VisaNet for The Most Wonderful Time of The Year
http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html, Oct 2013.

Bitcoin Wiki. Contracts: Example 7: Rapidly-adjusted (micro)payments to a pre-determined party.
https://en.bitcoin.it/wiki/Contracts#Example_7:_Rapidly-adjusted_.28micro.29payments_to_a_pre-determined

_party.

bitcoinj. Working with micropayment channels. https://bitcoinj.github.io/working-with-micropayments.

Leslie Lamport. The Part-Time Parliament. ACM Transactions on Computer Systems, 21(2):133â€“169, May 1998.

Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558â€“565, Jul 1978.

Alex Akselrod. Draft. https://en.bitcoin.it/wiki/User:Aakselrod/Draft, Mar 2013.

Alex Akselrod. ESCHATON. https://gist.github.com/aakselrod/9964667, Apr 2014.

Peter Todd. Near-zero fee transactions with hub-and-spoke micropayments. http://sourceforge.net/p/bitcoin/mailman/message/33144746/, Dec 2014.

C.J. Plooy. Combining Bitcoin and the Ripple to create a fast, scalable, decentralized, anonymous, low-trust payment network. http://www.ultimatestunts.nl/bitcoin/ripple_bitcoin_draft_2.pdf, Jan 2013.

BitPay. Impulse. http://impulse.is/impulse.pdf, Jan 2015.

Mark Friedenbach. BIP 0068: Consensus-enforced transaction replacement signaled via sequence numbers (relative locktime). https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki, May 2015.

Mark Friedenbach BtcDrak and Eric Lombrozo. BIP 0112: CHECKSEQUENCEVERIFY. https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki, Aug 2015.

Jonas Schnelli. What does OP CHECKSEQUENCEVERIFY do? http://bitcoin.stackexchange.com/a/38846, Jul 2015.

Greg Maxwell (nullc). reddit. https://www.reddit.com/r/Bitcoin/comments/37fxqd/it_looks_like_blockstream_is_working_on_the/crmr5p2, May 2015.

Gavin Andresen. BIP 0016: Pay to Script Hash. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki, Jan 2012.

Pieter Wuille. BIP 0032: Hierarchical Deterministic Wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, Feb 2012.

Ilja Gerhardt and Timo Hanke. Homomorphic Payment Addresses and the Pay-to-Contract Protocol. http://arxiv.org/abs/1212.3257, Dec 2012.

Nick Szabo. Formalizing and Securing Relationships on Public Networks.
http://szabo.best.vwh.net/formalize.html, Sep 1997.

Donate

Feel free to support this project by donating Bitcoin here:

```
1MUnEgxuMykj6z6rKnpozm8rCWpqdpSjen
```