cruzbit: A simple decentralized peer-to-peer ledger

asdvxgxasjab

For those interested, I want to elaborate on some of the details in the README found in the project's Github repository. Specifically, I want to highlight what makes cruzbit different from bitcoin (and other cryptocurrencies) and why certain design decisions were made in a bit more depth. I also want to discuss more of the philosophy behind cruzbit. This document could maybe be described as a "whitepaper."

Newer crypto

cruzbit uses the Ed25519 signature system. This isn't just for show or to be hip. There are substantial problems with ECDSA (the most common signature system currently used by cryptocurrencies) usage in practice. People have even written tools showing how some of the weaknesses can be exploited. The attacks are not theoretical. The upgraded signature system also protects against some classes of what are known as side-channel attacks. These may seem infeasible on the surface but they are quite real threats. They are one of the reasons why exchanges use Faraday tents and elaborate setups as part of their storage practices. The signatures are also just plain fast.

For hashing/proof-of-work I chose SHA3/Keccak. This also wasn't because the number is higher than SHA2 or for the simple sake of newness. In general, in cryptosystems newness for the sake of newness isn't always the best idea. If an algorithm is proven to hold up to scrutiny for many years it makes sense to continue to use it and not drop it for something new and shiny. But SHA3 is somewhat compelling. Its designers boast blazing fast hardware speeds which consume less power. In my opinion, you want a strong proof-of-work function to provide as much security as possible with the least thermodynamic cost. I see hardware mining as optimally efficient and if this is the most optimally efficient strong cryptographic hash function available, it makes sense, in my opinion, to use it. I'm not interested in designing-out the possibility of miner centralization. I'll let free market dynamics handle that problem.

Simplified transaction format

First I'll provide a quick overview of bitcoin transactions in order to make the comparison more clear. At a high-level, a bitcoin transaction has two critical sections. The first is a list of inputs and the second is a list of outputs. Some people refer to the outputs themselves as the "coins" in bitcoin. Working backwards, an output contains an amount and a piece of code ("script") required to be satisfied to redeem the amount specified in the output. The vast majority of the time, this code is effectively asking for the user to "provide the correct public key and signature to spend this output." An input is a reference to a previous output. It also includes the rest of the code required to satisfy the output's codified constraints, which is effectively just that the correct public key and signature are presented. The sum of the amounts of all of the outputs must be equal to or less than the sum of the amounts of the inputs. Any difference is considered an implicit miner fee. This means in addition to any scheduled subsidy, the miner of the block which confirms the transaction also can claim this difference.

In my opinion, this structure is somewhat awkward in practice and is a complexity that isn't entirely necessary. This puts a few requirements on a wallet (software which manages keys and creates and signs transactions.) It must know about all of the past transactions involved and how to properly construct these somewhat esoteric pieces of code to redeem their outputs. Already you've likely introduced the need for the developer to rely upon a bitcoin-specific dependency, which is sometimes difficult for a programmer to understand the fitness of. They just want to make a transaction. They don’t really care about output scripts or selecting inputs or any of these details.

This also means transaction sizes and verification complexity are not constant. Bitcoin code is littered with places that have to act on specific sizes in bytes and calculate signature operations to properly vet transactions for their validity. Not all transactions are equal in this respect which is why transaction fee calculation is transaction-context-specific. This is another fun detail a wallet developer needs to consider. As you can see, you are quickly requiring that a wallet developer be a near-expert in bitcoin protocol details. Being a good UI/UX designer and a skilled bitcoin protocol engineer is a large ask.

So back to cruzbit, to address the aforementioned complexities and awkwardness I got rid of inputs and outputs and those pieces of code/script. In a cruzbit transaction, there is a well-understood concept of a sender and a recipient. Both are basic Ed25519 public keys. Instead of code/script, there is just a simple signature. I saw little justification for the complexity of the script when most of the time people/entities transacting just want to move value from one key to another. If you have a good and easy to find SHA3 and Ed25519 implementation for your development environment (as the Go language has built-in here and here) you don’t need any other external dependencies to craft a validly signed cruzbit transaction. You don’t need *any* transaction history. You *do* need to know what the current height is of the cruzbit block chain (which number block is the most recent) but that and the balance of your public key(s) is the only context you really need to know.

In addition to these changes I also made the fee explicit. And in cruzbit a miner *must* claim all fees. It makes it nice for verification/sanity checking if all of the non-zero public key balances at a given block height match the scheduled issuance at the time of that block height.

There are other small changes to the transaction format mentioned in the

README but the above are the basics and most substantial. I'll discuss the most obscure field (but trivially easy to calculate), "series", in the next section.

No UTXO set

What is a UTXO set? In bitcoin, it's the set of all of the **U**nspent **T**ran**X**action **O**utputs. In cruzbit, there are no transaction outputs. There are only public key balances. In the previous section I discussed why I think that is more simple and safer for wallet developers to deal with. But I also think it is more conceptually simple for all users to grasp. We're used to having an account and for it to receive credits, and unfortunately debits. In cruzbit you can think of each public key as a mini-account.

Cruzbit is not the first cryptocurrency-style ledger to abandon the UTXO-model for the account-model, however, as far as I'm aware, all of those which have moved to the account-model have done so in roughly the same way. They have introduced an abstract notion of an account. Further, this account's transactions must all include a nonce. But this nonce isn't pseudo-random. It is a serial nonce which *must* increase by 1 for each transaction the account generates. The wallet must remember this nonce (or retrieve it from a network node) and use care to craft transactions in the correct order. Any break in the sequence disrupts the processing ability of subsequent transactions. All nodes in the network must also remember the most recent nonce used by all accounts.

That sounds a little complicated. Why does this per-account serial nonce exist? While inputs and outputs are a bit unwieldy, in my opinion, that model has some nice properties. The developer of bitcoin wasn't being weird for weirdness' sake. It was a calculated decision. With inputs and outputs you don't have a problem of transaction replayability. In short, replaying a transaction means resubmitting an already processed transaction to the network in order to redeem the funds twice. By chaining transactions, the input and output model has this protection built-in. No two valid transactions will ever look identical. Whereas public key *P* paying public key *K* 2 cruzbits could very well look the same as another transaction in the future which involves the same keys for the same amount. There needs to be a way to differentiate them, hence a nonce. Now that answers the question of why does an inputless/outputless transaction need a nonce but why does it need to be serial and per-account? If it weren't serial, network nodes would have to keep track of *every* nonce ever used by each account and make sure any new transactions don't contain re-used nonces. That requires much more storage than a simple serial per-account nonce. The savings should be clear.

But then why *doesn't* cruzbit require a serial per-account nonce? Cruzbit differentiates transactions by including a traditional pseudo-random nonce. If nonces aren't re-used per-sender no two transactions should look the same. But cruzbit doesn't track these nonces. It only tracks which transactions have been processed by their hash. It avoids having to remember the hash of every historical transaction by introducing a network-wide "series." The series is effectively a serial nonce that the whole network uses for a period of time. That period is 1008 blocks (roughly 1 week.) The calculation for the correct series to use at a given height is:`current block chain height/1008 + 1`

Transactions have a grace period. At any given block height the current and previous series are acceptable when submitting a transaction. This exists to mitigate potential issues arising from transaction queuing delay and other potential effects of the time frame being too rigid. Humans aren't very good with strict deadlines. But this all means cruzbit network nodes can "forget" about transactions older than the previous series for the purpose of processing new transactions. It's a trade-off. I think it's a conceptually simple one which places much less burden on users but perhaps it's debatable. The cost is network nodes have to always store the last two series' worth of transaction hashes vs. having to store every account's most recent nonce. I think in practical terms the storage costs are probably effectively equal but this can also be debated.

The other nice property of the UTXO model is that in order to verify new transactions, network nodes only need to know about the set of unspent transaction outputs. In cruzbit, nodes need to remember the balances of all public keys with a non-zero balance. The storage requirements of these two sets are *probably* roughly equal in practice. But in cruzbit, nodes also need to store the hashes of the last two series' worth of transactions. This is a marginally larger storage requirement for the benefits of not having to deal with inputs and outputs and their aforementioned associated complexities. I think it is a clear win but it is likely debatable.

No fixed block size limit

Due to inputless/outputless transactions being roughly the same constant size, it isn't necessary to restrict blocks by size. We do it by transaction count. But this restriction isn't fixed. It moves with "piecewise-linear-between-doublings growth." You can read more about it in BIP 101. That's basically what cruzbit has adopted with some slight deviations. In my opinion, it's unreasonable to restrict block sizes to a single size forever. It doesn't make rational sense. Computing and storage capabilities grow as does healthy network usage. Debating the growth factor of blocks in a block chain network is a painful experience and I think it is much easier if it's baked in to the protocol from the beginning. In that sense everyone knows what they are getting into upfront. No tricks nor surprises. With BIP 101 we know the maximum capacity of the network at any future block height and can plan for it accordingly. It is also a very simple solution. Miners can restrict the sizes of blocks they choose to create and build off of. I don't think they also need control over the maximum acceptable size for the entire network so I am against explicit miner controls for this and I am also against complexity arising from more dynamic models. Both of these alternative approaches also negatively impact long-term predictability. I guess we'll see how it works out in practice.

Reference implementation is in Go

In my opinion, Go is a much easier language to read and work with than C++. It is also very popular right now and has a lot of new developer interest. In some ways, I think C++ is a bit esoteric, which is an accusation I also level on the bitcoin protocol. I considered using Rust but I'm concerned about the learning curve for new developers. I want the reference implementation to be as understandable as possible for the largest number of developers. But do be on the look-out for a future Rust port of the cruzbit client.

Web-friendly peer protocol

All peer communication is with secure WebSockets. By secure here, I mean that the channel is encrypted with TLS and the `wss://` protocol scheme is used. In the current default configuration there is no protection against man-in-the-middle attacks because the certificates and keys used are ephemeral and not signed by any publicly acknowledged certificate authority. I chose WebSockets because I wanted to use something standard and bi-directional. Again, I didn't want a user to have to use an obscure library nor understand low-level TCP socket programming subtleties. I think all modern development environments have a WebSocket implementation at their disposal so that made it a natural choice. While building cruzbit, I often used Chrome's Javascript console to connect to the client and debug the protocol.

The actual protocol and all primitives are structured in JSON. This is probably a controversial choice. But again, I didn't want to require obscure encoding schemes or constructions not readily available to all development environments. And maybe most importantly, I wanted humans to be able to work with it easily. Block header hashes and transaction hashes are computed using their strictly-serialized JSON forms. This means no white-space and strict ordering of fields. I say this choice is controversial because JSON encoders/decoders are notorious for having finicky compatibility issues across environments. I'm hoping our formats are rigid enough that this isn't a material concern in practice.

So how is it like bitcoin?

The consensus rules around block processing are otherwise unchanged. There is a target value of which the block header's hash must be less than when compared as 256-bit integers. Every 2 weeks (2016 blocks) there is a re-target of this value designed for new block creation to converge on an average 10 minute interval depending on network hash-power. The new target is calculated the same way bitcoin's is (except modified to include protection against the "Time Warp Attack"). The most-work chain wins and each block must build off of a valid predecessor in the chain. The subsidy schedule is also identical, every 4 years (210000 blocks) the block reward halves. And no more than 21 million cruz will ever exist. One other odd shared quirk is that new mining rewards are subject to a 100 block maturation before they're applied to a miner key's balance. It is for the same reason, which is to avoid an unpleasant UX in the event of an honest block chain reorganization.

Philosophy

I designed cruzbit to function as a store of value and a transaction base layer. I think it makes sense for any base layer to be as simple and basic as possible. This is what I have tried to achieve. I have no political nor ideological motivations and I am only interested in trying to provide such a base layer. I believe bitcoin currently is this base layer but I believe it has its quirks and short-comings which I hope I'm addressing. It's possible I'm not and this attempt is misguided but I think there is only one way to find out. I'll let the market decide.