An Introduction to the Zen Protocol

Zen Protocol Development

October 16, 2017

Summary

Decentralized platforms let their participants avoid counterparty risk without using trusted intermediaries. The Zen Protocol is the basis for such a platform. Zen is a *parallel blockchain* to Bitcoin, allowing users to create assets that react to events on the Bitcoin network. Zen targets financial use cases such as trading and creating new kinds of assets. In this white paper, we discuss how Zen works, how it differs from existing smart contract platforms, and what sort of applications it makes possible.

Contents

List of Figures

1 Introduction

Bitcoin was designed to be digital money without central authority. It succeeded by identifying the key problem—consensus on ownership—and providing a mechanism to reach consensus that cannot be subverted without using real, costly resources. This solution, a *proof of work blockchain*, has since been widely imitated.

We introduce Zen, a blockchain to trade in any kind of financial asset with the same consensus on ownership and freedom from central control that Bitcoin provides for money. These assets connect to digital agreements with powerful, programmable abilities to process information and grant rights. Zen observes the Bitcoin network and allows for assets backed with Bitcoin.

Before discussing Zen in detail, let's look at what can be done with the Bitcoin protocol itself.

2 Digital agreements in Bitcoin

Beyond consensus on transfers of money from one owner to another, Bitcoin also permits consensus on more complicated kinds of digital agreements. For instance, multi-signature schemes allow bitcoins to be locked with more than one digital key. Other digital agreements move part of the consensus off the transaction ledger entirely, using decentralized consensus only to give each party the ability to defend against a broken agreement. Bitcoin's *Lightning Network* uses agreements of this kind.

These sorts of digital agreement have become known as *smart contracts*. Smart contracts on Bitcoin have three distinctive properties:

1. They relate to the flow of bitcoins themselves, i.e. money, rather than other assets.

2. They consume network resources otherwise used for the direct transfer of money.

3. They are only necessary for applications which have to be free of central control.

Bitcoin supports smart contracts just as well as is needed to improve Bitcoin's ability to be decentralized digital money. To this end, Bitcoin's scripting language is intentionally restrictive, preventing contracts from consuming excessive resources, and requiring them to be extremely efficient. If possible, Bitcoin smart contracts pay their own way by replacing transactions which would otherwise have to be recorded on the blockchain.

For assets other than Bitcoin, and for contracts which require more than a minimal amount of computation to validate, Bitcoin does not provide the right solution for reaching consensus. These uses compete with Bitcoin's *monetary function*—secure, reliable digital money without central control. This conflict is not particular to Bitcoin. **Any blockchain which attempts both to create money and to provide smart contracts must limit one or the other.**

3 Design of the Zen Protocol

The Zen blockchain is designed to support real financial products, with powerful automation and freedom from central control. These uses motivate many decisions about the Zen Protocol's architecture. This section gives a high-level overview of that architecture.

3.1 Bitcoin integration

Zen has a scarce native token for contract activation, but **this token is not intended as monetary competition**. Rather, Zen supports tight integration with the Bitcoin blockchain, allowing for Bitcoin-backed assets that can be used as currency on the Zen Protocol. The scarce native token means that there's no need for an "official" Bitcoin-backed asset[1]: *multiple, competing* services put Bitcoin-value onto the Zen blockchain. At the same time, the native token lowers the cost of using Zen by subsidizing the miners who secure the chain. Every Zen full node keeps track of Bitcoin consensus, giving contracts the ability to use the state of the Bitcoin blockchain when deciding what actions to take. The Zen Protocol attempts to be forward-compatible with updates to Bitcoin, such as drivechains, that offer a decentralized way to move Bitcoins between different blockchains.

This solves the problem of smart contracts competing for the resources used to secure money transfer.

3.2 Verified contracts

Contracts come with proofs about how long they take to run—which means they can be compiled and run much faster—and can come with proofs about what they do—which means that other contracts can understand them and treat their assets accordingly. All these proofs are written in the same language as the contracts themselves. Section 8.2 explains how these proofs work.

Zen contracts **never** cost the user resources unless they execute **successfully**—there are no "out of gas" errors.

[1]See Rootstock, discussed below, for a blockchain that *does* have an "official" asset.

3.3 Contract lifecycle

In Bitcoin, the full code of a “contract” is recorded in the blockchain each time it is used. In Ethereum, contracts are created once, and then go on to live in the blockchain indefinitely. Zen contracts are *pay-per-block*: they pay miners for every single block in which they are active. This is a miner-friendly and node-friendly solution to persistent contracts: only the contracts that can actually affect the blockchain are cached in memory, and miners get paid for caching them.

3.4 Chain security: Multihash Proof of Work

Zen uses proof of work to secure the blockchain’s transaction history. Proof of work demands that miners invest real, unforgeable resources. The commonly proposed alternative, proof of stake, has not been demonstrated to combine security, open access and incentive compatibility. In particular, the “stake” of “proof of stake” is the same asset with which the stakers are compensated. This entrenches existing owners of the asset at the expense of future users. Proof of stake also couples the security of the blockchain to the value of a particular token in the blockchain: if the token changes in value, the cost of rewriting the blockchain also changes. This is not the case for proof of work, where the cost of rewriting the blockchain is determined by the *physical* resources that were used to secure it.

Miner centralization describes the phenomenon of mining power becoming concentrated in the hands of a few individuals or companies. This is a real concern for a decentralized platform, and has even become a concern for Bitcoin. Zen reduces the influence of a single powerful miner by introducing **multihash mining**, described in section 5 and in Perlow and Cook (2017).

3.5 Assets

The idea of *owning an asset* has the same basic meaning in Zen as in Bitcoin: it means being able to “unlock” that asset with a digital signature. Zen uses standard public-key authentication for user-owned assets.

Zen supports multiple assets types at the protocol level. That means that all contracts can understand and use all assets, and that assets can be held and moved with simple, efficient public-key authentication. It lets assets be used in a Zen-based *Lightning Network* and makes it easy to swap them with assets held on other blockchains. Contracts make their own assets and define their uses.

3.6 Light client security

In a decentralized smart contract platform like Zen, it’s unlikely that users on less powerful devices like mobile phones will download and fully validate the entire blockchain. This makes it very important that these users still be able to obtain a good level of security, without trusting a centralized source of information. Bitcoin's solution, *Simplified Payment Verification* or *SPV*, uses proof of work "on top of" a transaction. Zen does the same, including additional *commitments*—information that light clients can use to prove additional properties, such as whether a contract is active or not. Zen uses data structures like the efficient sparse merkle tree that are fast to update and that make proofs fast to generate.

3.7 State control

All blockchains have state, and give transactions access to some of that state. One simple example is that each transaction "knows" whether the value it transfers actually exists or not! The Zen Protocol controls where state can live: with limited exceptions, the only mechanism is to put data in transaction outputs. This is similar to Bitcoin, and distinct from protocols like Ethereum and Tezos, which allow contracts to store data in a global store. Contracts in Zen are immutable. They can store their state in transaction outputs, but their source code never changes.

This design makes it easier to reason about smart contracts. Unlike protocols in which rights are administered via data areas with so-called "private" access rights, the Zen Protocol makes it clear that rights should be granted either by cryptographic authentication, or by possession of tokens.

4 Other platforms

Apart from Bitcoin, which restricts smart contracts to those which move bitcoins and don't compete with Bitcoin's own monetary function, other blockchains have offered smart contracts[2]. We look at a few of them to see how they compare with the Zen Protocol's solution.

4.1 Ethereum

Ethereum tries to provide both a currency, Ether, and a platform for decentralized applications, or *dApps*. It implements these applications in a single "world computer". Assets are implemented in this world computer *as a kind of application*, at a higher level of abstraction than Ether itself. This means that in Ethereum, all transactions must be paid for in Ether. Contrast this with Zen, where only contract creation requires the native token—a normal transaction fee can be paid with any asset. Using the internal currency as the unit of account creates the conflict we mentioned above: the dApps make

[2]Non-blockchain solutions like Open Transactions are also available. We limit ourselves to discussing platforms which purport to be free of central authority.

moving Ether more expensive, and the monetary function makes the dApps more expensive.

Without first class assets, doing anything with an Ethereum token means running the dApp that created it—even sending it to another user. The dApps themselves can't understand new asset types without a trusted human in the loop to verify they're "real" tokens.

Ethereum intends to move to the so-called "proof of stake" system for determining consensus, retiring proof of work. While proposed as a potential efficiency measure, it is not clear how this could prevent contention for resources between the monetary function and smart contract capability.

Ethereum's contracts are notoriously hard to verify as correct. This is not a fault particular to Ethereum—"formal verification" is often difficult—but Ethereum was not designed to make verification easy. This extends to proofs about how long contracts take to execute. When these proofs exist[3], they're useful to measure how much gas is necessary, but don't fix the fundamental problem that the gas system creates: slow, interpreted contracts that can fail in the middle of execution, while costing the user money.

The gas system creates another problem, which formal verification cannot solve: if users give conflicting instructions to a contract, then only one can win the race, but all the users must pay for making the transaction. What's worse, miners are incentivized to re-order transactions so that the one which pays the most gas wins.[4] This particularly affects applications like auctions, markets and capped token sales.

4.2 Rootstock

Quite simply, Rootstock is intended to be Ethereum with bitcoins instead of Ether. This involves using a *sidechain*—a blockchain which connects to Bitcoin, enabling users to move bitcoins on and off the sidechain. Rootstock's smart contract logic and internal architecture are almost identical to those of Ethereum, with the same shortcomings.

As a sidechain, Rootstock goes some way towards separating monetary functionality from smart contract capability, but we consider it unlikely that it will solve the problem, for two reasons. Firstly, by adopting Ethereum's architecture, Rootstock demands that the user pay for every single transaction with Rootstock-bitcoins, just as every Ethereum transaction must be paid for with Ether. This means that the more smart contracts are used, the greater the impact on the monetary supply, the ease of sending money between people, and so on. Secondly, the cost of securing Rootstock's blockchain is paid entirely in transaction fees, meaning that short-term fluctuations in the

[3]Bhargavan et al. (2016) demonstrate gas-cost proofs by translating Ethereum contracts to F*.

[4]This scenario—everyone pays to bid, only one person wins—is known as an *all-pay auction*.

overall demand for smart contracts have an immediate effect on the security of the platform.

Rootstock has stated its intention to move to a decentralized sidechain model when the Bitcoin protocol supports a suitable mechanism, such as drivechains. However, in its current form, Rootstock has a central authority. Rootstock's mechanism for moving bitcoins between chains is a single, trusted "federation". In the federation model, a group of companies acts as depositories, holding the bitcoins of everyone using the sidechain, and releasing them according to the rules of the Rootstock protocol. As a key purpose of a blockchain is to remove the need for trust in a single authority, this makes the blockchain itself unnecessary!

4.3 Tezos

Tezos is a clean-sheet design for a blockchain. Intended as "the last cryptocurrency", it aims to supplant both Bitcoin and Ethereum by providing money and performant smart contracts, with a decentralized mechanism for upgrading the blockchain itself[5].

As a single-asset blockchain which competes for the monetary function, Tezos suffers from a conflict with smart-contract functionality. Unlike Zen, it will use proof of stake. However, it shares with Zen the goal of making it easy to prove things about contracts.

Tezos uses a custom stack-based language called Michelson. However, the *proofs* are not written in Michelson—they will be written in existing theorem proving systems like Isabelle or Coq. Apart from requiring programmers to learn more than one new language, this means that the Tezos protocol itself will not be able to use these proofs initially[6].

Tezos does not even attempt to prove how long contracts take to execute. It simply takes the brute force approach of putting a global limit on execution time. This means Tezos contracts must be interpreted, not compiled, and guarantees either that quick contracts will be too expensive or that slow contracts will be too cheap.

5 Multihash mining

For a full overview of multihash mining, see the paper.

Back to Zen. This section discusses a new way to secure the blockchain with proof of work: Multihash mining.

Proof of work ties blockchain integrity to real work. As occurs in almost any sort of work requiring capital investment, miners benefit from economies

[5] Users submit patches written in OCaml, which coin-holders then vote on.

[6] It might still be possible to write a "Coq-parsing patch" in OCaml and submit it as a protocol upgrade. We consider this... inelegant.

of scale, which can lead to fewer independent miners. This miner centralization introduces several risks, one of which is increased conflict between miners and users.

To mine efficiently, each individual miner must *specialize*, buying equipment which is particularly good at mining under some particular hash function or category of hash functions. For instance, Bitcoin miners use ASIC-based hardware which can only mine under the `SHA-256^2` algorithm. Litecoin miners use ASIC hardware which can only mine under the `scrypt` function. Meanwhile, some hash functions do not yet have specialized hardware—miners normally use GPUs or CPUs with these functions.

Multihash mining takes advantage of the natural economies of scale implicit in proof of work, using them to reduce the risk of miner-user conflict.

5.1 Multiple hash functions

Each block in Zen has a proof of work attached. The Zen Protocol accepts any of several different kinds of proof of work, each corresponding to a different hash function. Each hash function has its own difficulty: nodes verify blocks by checking the proof of work against the difficulty of the hash function used.

Without any additional changes to the basic proof of work algorithm, this leads to the difficulties becoming unbalanced over time. Unpredictable events, like different rates of improvement in hardware, or changing profitability on other blockchains, make it relatively too easy or too hard to mine on a particular hash function. The Zen Protocol resolves this by targeting a *hash function ratio*. The number of blocks mined under each hash function is counted over a two week period, and the difficulty of each hash function is adjusted to create more or fewer blocks over the next period.

5.2 Token holder influence

The hash function ratio is not fixed: owners of the Zen native token can change it over time. Every two weeks, each owner of the Zen native token can vote on which mix of hash functions will be used. If the mix shifts towards a particular function, that function's difficulty is reduced, making mining easier and increasing the reward for its miners. If the mix shifts away, then a difficulty increase reduces the reward for the miners using that function.

Because miners specialize in some algorithm or algorithms, this vote has a real effect on their profitability. However, miners are not affected as quickly as they would be in the case of a hard-fork to change the proof of work, and the effects of the change can be reversed with the next vote. We expect that this "iterated game" will encourage miners and token holders to cooperate.

6 Transactions

In the Zen Protocol, all value is stored in "unspent transaction outputs", or *UTXOs* for short. UTXOs have two parts:

1. the value they store,
2. the conditions for unlocking that value.

The value in a UTXO is simply a quantity of some asset/token. (Zen has many different kinds of token.) A condition for spending a UTXO is called a *lock*. There are two classes of lock:

Simple locks correspond to very common use-cases. One example is the `PKLock`, which does the same thing as pay-to-pubkey-hash (P2PKH) in Bitcoin. For example, to spend a UTXO with the lock `PKLock BCFF...3A12`, a transaction must contain a signature for the public key with hash `BCFF...3A12`.

Contract locks can only be unlocked by a contract. A UTXO with the lock `ContractLock 65B9...47C4` can only be spent by the contract with identifier `65B9...47C4`.

This makes transactions similar to those in Bitcoin—they unlock inputs, and lock the value to new "outputs". Signatures, information for contracts etc. all go into a special "witness" field. There is no equivalent to Bitcoin's script language—contracts handle all the complex logic.

Witnesses don't affect the hash of a transaction. This means that transactions in the Zen Protocol are protected from malleability attacks—the same benefit which Bitcoin gets from its SegWit upgrade.

Users don't see things like `PKLock BCFF...3A12`. For convenience, addresses are encoded with error protection and a short identifier—Zen uses Bitcoin's new `Bech32` format (Wuille, 2016).

6.1 Why UTXOs?

The two common paradigms for blockchains are *UTXO-oriented* and *account-oriented*. In the UTXO-oriented paradigm, assets are stored on outputs, which are destroyed as soon as they are spent. Think of this as taking some coins, melting them together, and casting new coins: the amount of metal in the new coins has to be the same as in the old, but the individual coins may look different, be marked with different owners, and be of different sizes. We refer to the UTXO-oriented paradigm as *coin-oriented* to emphasize the importance of the coins themselves.

In the account-oriented paradigm, assets live in a particular account, and "sending an asset" means *removing* an asset from one account and putting it *inside* another account. The account-oriented paradigm is slightly more intuitive, but the UTXO/coin-oriented model has several advantages:

Non-blocking transactions

Naive account-oriented protocols suffer from a *replay attack* vulnerability, by which valid transactions can be used more than once. For instance, after taking payment from someone, an attacker can reuse the same transaction to drain the buyer's account.

To prevent these vulnerabilities, account-oriented protocols require that each transaction from an account include a counter that increases by one every time the account is used. This forces transactions to occur in the blockchain in the same order in which they were created, with no gaps. Unfortunately, this leads to poor performance under load: each transaction blocks any further transactions from that account until it is either mined or dropped from miners' mempools. Account-oriented protocols encourage the user to keep only one account: any user who does so can be completely blocked from using the blockchain until the network load event is over.

Coin-oriented protocols are not susceptible to this vulnerability: attempting to reuse a transaction means attempting to spend the same coins, which is impossible. With no transaction counter, coin-oriented protocols usually perform better under load.

Parallel validation

In Bitcoin, any transaction can be validated with very limited data: knowing that its inputs are unspent, together with the current block height and timestamp, is enough. The *effects* of a transaction are simply that the inputs are consumed and the outputs are created. This makes it possible to validate most transactions independently of each other. Zen transactions can require a little more information, and have more effects, but most transactions do not affect each other's validity, and it's fast to check which ones do. Account-oriented systems usually need strictly linear validation, as the order of transactions can have arbitrary, hard to check effects on validity.

State management

Coin-oriented protocols make it natural to use contracts where all or most of the state—data which changes over time—is stored attached to the coins themselves. This makes it very clear how different contracts can affect each others' state, and gives them full control over how they manage their own state. In practice, many digital agreements don't require any state at all—these agreements can simply not create any state-carrying coins.

7 Tokens

Unlike Bitcoin, which has only one kind of token, and protocols like Ethereum, which use custom contracts to implement some of the functionality of tokens, the Zen Protocol's tokens are **first class citizens**. That means that every sort of token in Zen has a similar status to the Zen native token. Tokens are stored in transaction outputs, just as in Bitcoin, and can be unlocked with the right permissions, then locked again in new outputs.

7.1 Moving tokens: a comparison

In Bitcoin, the value of a transaction's output represents a certain number of bitcoins locked in that output. Bitcoins are spent by unlocking outputs, then locking its bitcoins inside new outputs. For instance, this transaction takes an output with five bitcoins, another output with two bitcoins, unlocks them both, and locks them in two new outputs:

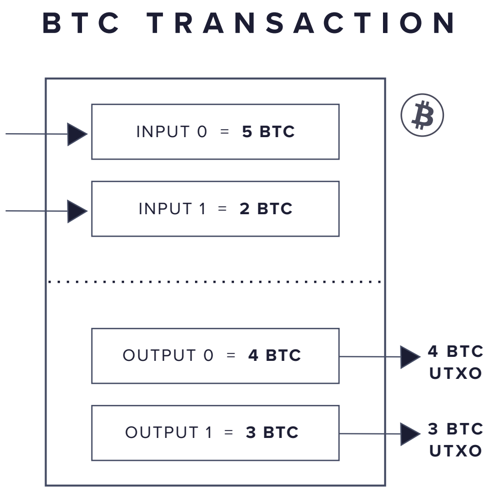


Figure 1: Bitcoin transactions unlock inputs, freeing the bitcoins inside—then immediately lock those bitcoins inside new outputs.

The two new unspent transaction outputs (*UTXOs*) are then available to be spent.

In Zen, transaction outputs can lock tokens of any type. This transaction unlocks a UTXO carrying two of token X, another UTXO carrying three of token Y, and a third carrying five of token X (the same type as the first UTXO), then locks them to some new outputs:

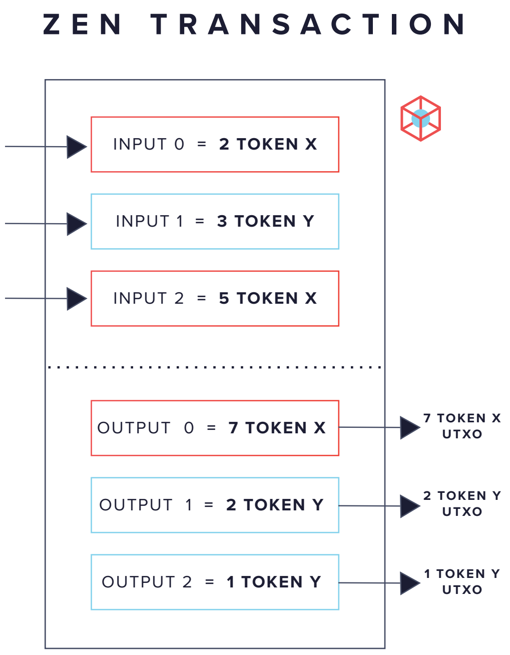


Figure 2: Zen transactions can unlock any kind of asset. They immediately lock those tokens to new owners, inside new transactions outputs.

Just as for Bitcoin, the new UTXOs can be spent according to their new conditions, or locks.

7.2 Different types of token

Each contract can create 2^{256} kinds of token. There are almost 2^{256} possible contract IDs, so in total, there are almost 2^{512} possible kinds of token.

Zen also has a single asset which is *not* issued by a contract. This is the native, mined ZEN token. We discuss the need for this token in section 8.1.1.

7.3 Using tokens

First class tokens provide a convention for representing financial rights. Both contracts and users can immediately see which assets they hold—without passing messages to other contracts, parsing contract code, or using a trusted source of information about assets. This makes it easy to create powerful digital agreements: for instance, a contract can accept any kind of asset as collateral, then issue a token acting as a option on that asset, without knowing anything about the asset itself. Users who own this contract's options can see them in their wallets without any upgrades, add-ons, etc.

Zen's token support also makes it easy to *sell and transfer rights*. Consider the example of an interest rate swap. This is a popular financial product in which two parties agree to pay each other interest payments on some lump of capital. One party agrees to pay at a fixed interest rate, while the other pays at a floating (market) rate of interest. In Zen, this digital agreement would use a contract that issues two different kinds of token, one representing the fixed rate, the other the floating rate. The contract also holds a reasonable amount of collateral from each party, in order to cover the expected payments.

Suppose that the party which pays a fixed rate wants to close his position. Once he finds someone willing to buy him out, it takes only a single transaction to exchange the payment for the token which grants the right to the "fixed leg". This transaction happens without interacting with the interest rate swap contract.

8 Contracts

Contracts and users have similar abilities in the Zen Protocol. Both interact with the blockchain by creating transactions. Contracts and users can even co-operate to create a single transaction—the basis of "level 2" protocols. But whereas users are pseudonymous, and use public key cryptographic to control their assets, each contract has a unique identifier, and automatically has control over assets locked to that identifier.

8.1 The Active Contract Set

Contracts live in the *active contract set*. Only contracts in this set can create transactions and interact with the blockchain (Figure 3). Contracts enter the ACS when a user pays a *contract sacrifice* in the Zen native token, part of which goes to the miner who activates the contract, with the rest going to all the miners over the contract's lifetime. Contracts can be *extended* by additional contract sacrifices. When their sacrifices run out, they leave the ACS, becoming *inactive* until someone reactivates them by providing source code and a new sacrifice.

8.1.1 The Zen native token

The payment for activating a contract is unusual: it doesn't just go to the miner that records the activation. Whenever a user makes a normal transaction, two things happen:

1. a miner includes the transaction in a block, and

2. that same miner receives the transaction fee.

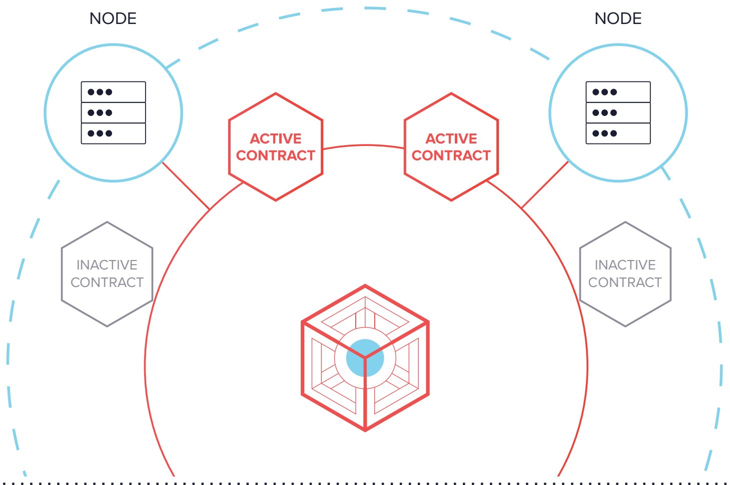


Figure 3: Only active contracts (marked in red) can affect the Zen blockchain. Nodes remember active contracts, but don't need to store inactive contracts.

These two actions—including a transaction in a block, and receiving its fee—are linked together: either both happen or neither does. This gives each miner a simple choice: does the transaction fee make it worthwhile to include the transaction? On the other hand, when a user activates (or extends) a contract, the following happens:

1. a miner includes the transaction in a block,
2. that miner receives the transaction fee,
3. a contract becomes active for a certain number of blocks,
4. each miner of those blocks has to track that contract in the ACS, and
5. each miner of those blocks is paid part of the transaction's contract sacrifice.

Only the miner who first includes the transaction in a block has any choice about activating the contract. The other miners have to keep track of the active contract, but don't get to decide on the transaction fee (which in any case goes to the first miner), or the contract sacrifice (which has already been determined). That makes it important to reward these other miners with an asset that we know they want.

For this reason, the Zen Protocol uses the native token, generated by mining, as the asset used to activate contracts. This asset is a *focal point*: all miners are already interested in mining for it, so all miners are likely to want contract sacrifices paid in the same asset.

8.2 Resource verification

All contracts in the Zen Protocol prove how long they take to run before they ever enter the blockchain. The proofs are checked automatically by each node. The Zen Protocol enforces this by using a dialect of the F* programming language to implement *cost types*, which verify what resources a program uses at compile-time (Figure 4). The F* language is part of consensus: the source code of each contract goes directly into the blockchain.

This makes Zen contracts efficient: each node can compile a contract, and then run it as many times as needed, without counting resource usage.

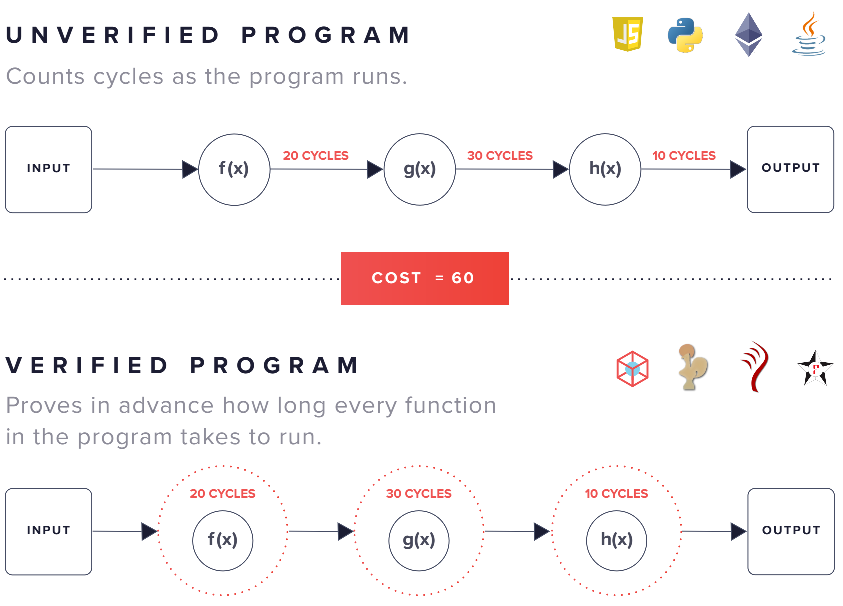


Figure 4: In conventional “Turing-complete” smart contract platforms, an interpreter measures the cost of running a contract, every single time it executes. In Zen, each part of a contract is wrapped in a proven cost. Zen’s contract language uses these costs to verify the cost of the whole contract, before it ever runs.

What do contracts and proofs look like? Some languages, like Viper, are intentionally restrictive, making it impossible to write programs whose running time can’t be automatically calculated. Other languages are more flexible, but need explicit proofs, which can be hard to find.

The F* programming language is powerful enough to write programs in a natural style, but also provides an assistant which can find some proofs about these programs automatically. That means that most of the time, developers just write the contract, and leave it to F* to calculate the running time. When developers do write proofs, they only have to write them about the small pieces of the contract that F* can't prove by itself.

8.2.1 On being Turing-complete

As Alan Turing proved in 1936, there are programs which cannot be proven either to halt or to run forever. As Zen requires that contracts halt, and in fact a proof of how long it takes for them to halt, Zen is not Turing-complete—that is, not all possible programs can be used in Zen's smart contracts. But in practice, we rarely use algorithms for which we don't have a good idea of how long they take to run. Decentralized platforms in particular have quite limited resources for doing computation, and allowing programs to run indefinitely would just break consensus between users.

Even when we do use algorithms that might not halt, we eventually have to stop the execution of those algorithms. Usually we can do this automatically, by monitoring how many times some outer loop of the algorithm has been used. Only in rare cases is the internal behaviour of an algorithm completely chaotic.

8.3 Formal verification and contract communication

Zen's contract language is powerful enough to use *formal verification* to do a lot more than check resource usage. Contracts can prove things about the assets they create, and other contracts can use those proofs. For example, the interest rate swap contract in subsection 7.3 can prove that its assets are collateralized swaps that use the *LIBOR* interest rate. Other contracts can then identify those assets and use them in their own portfolios.

8.4 How contracts work

Contracts only change the state of Zen's blockchain by creating transactions. Each contract implements a function that takes some of the existing blockchain state—unspent transaction outputs, block headers, and information about the Bitcoin blockchain—and returns a transaction. Nodes verify transactions by running the contract to see if the output is the same.

Each contract is *sandboxed*: to communicate with other contracts, it must create a transaction that carries a message. Contracts are also *immutable*: they can manage their state in UTXOs, but their code never changes. As mentioned in section 6.1, these two properties give contracts full control over their own behaviour, and make it harder to create many of the vulnerabilities that have caused problems in existing smart contract platforms.

8.5 Managing assets

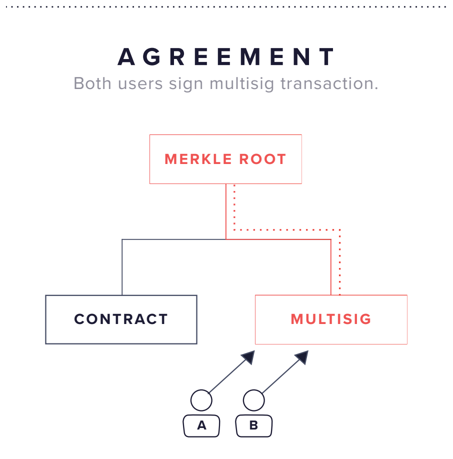
In subsection 7.2, we mentioned that each contract can create many different kinds of token. These tokens are tagged with the contract that creates them—contract X can't create a token tagged with contract Y. However, *contracts can use any kind of token*—both those issued by other contracts, and the native Zen token, which is created by mining. Contracts can see what assets they own by looking at the UTXO set.

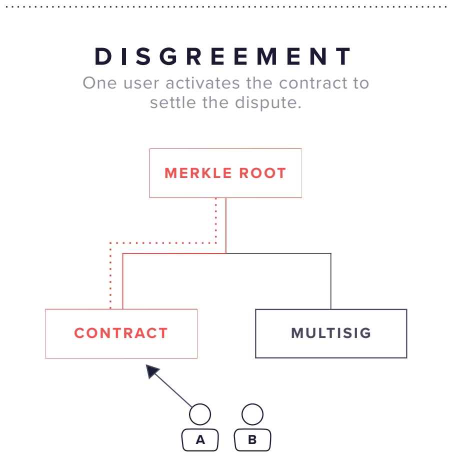
8.6 Off-chain contracts

It's possible to lock assets to a contract that has never been activated. To do this, a user checks that a contract is valid, and then, without publishing it, works out its identifier by taking the hash of its source code. This produces a contract address like `c67S4...LdN3` which can be used like any other contract address. The contract can then be kept secret until activated: the assets can only be spent by revealing the contract's source code.

8.7 Confidential agreements and decentralized escrow

Not all agreements should be public. Private agreements are possible using a simplified form of the *MAST* proposal for Bitcoin (see Lau, 2016). In this scheme, the parties to an agreement take a few different conditions and turn them into a tree, then use the merkle root of that tree as their contract identifier (Figure 5). Using the contract means revealing a single branch of the tree, then using the condition at the tip of that branch. The "subcontracts" of the agreement are only published to the blockchain if they are used to resolve a dispute. If the parties all cooperate, they can use a multisig branch of the tree—keeping the terms private.





Both parties agree on the outcome of the agreement. They both sign a multi-sig transaction, then use a proof of inclusion (represented by the red path) to show that the multisig conditions can be reached from the merkle root. The contract stays secret.

The parties disagree. Party **A** activates the contract, and uses it, together with a proof of inclusion, to enforce the agreed terms. **A** gives up confidentiality to use decentralized enforcement.

Figure 5: A confidential digital agreement between two parties.

9 Bitcoin

Existing blockchains relate to Bitcoin in one of two ways. *Alternative* blockchains, or *altcoins*, compete to provide a new variant of money. *Sidechains* depend completely on their parent chain. Zen takes a third approach.

Zen is a *parallel* blockchain. Zen tracks the whole Bitcoin blockchain, recording Bitcoin block headers. Miners compete to include Bitcoin block headers in Zen blocks. A few days later, Zen nodes reach consensus on whether these headers ended up in the best valid Bitcoin blockchain. The miners are then rewarded for these blocks. (See Figure 6.)

This *delayed consensus* mechanism avoids most of the problems with chain reorganizations that sidechains experience. The delay means that short Bitcoin reorganizations don't have any effect on the Zen blockchain[7]. Contracts which need high security can use the validated best Bitcoin chain to decide what actions to take.

At the same time, contracts can still see more recent Bitcoin headers. Contracts which don't need as high a level of security can use *reorganization insurance*: a Zen asset class that pays insurance when the Bitcoin chain reorganizes.

[7]Reorganizations of more than a few blocks have been rare in Bitcoin. None are known to have altered more than a few hours of the ledger's history.

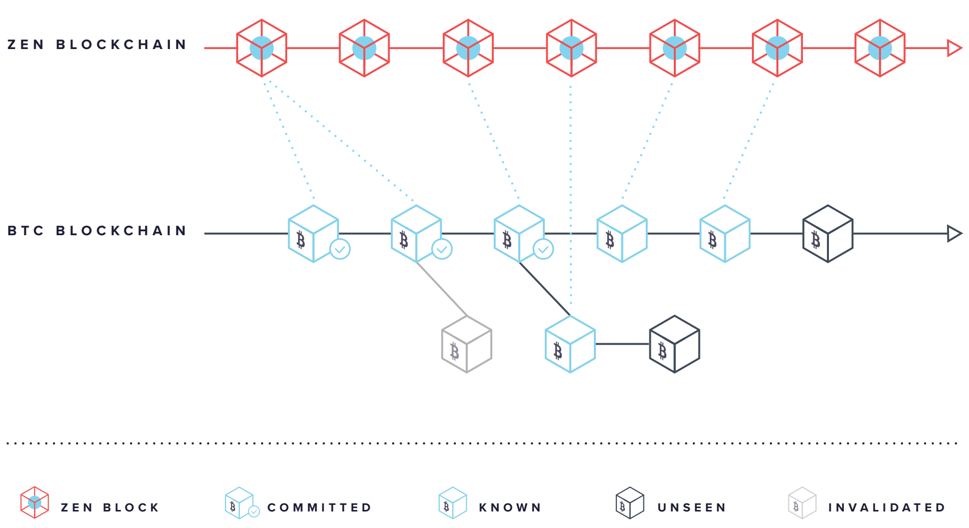


Figure 6: Zen nodes track both the Zen blockchain and the Bitcoin blockchain. Zen blocks contain Bitcoin block headers, making the Zen Protocol aware of what happens in Bitcoin. In this figure, the Zen blockchain knows the headers of all the blue-coloured Bitcoin blocks, but only the three oldest, marked with ✓, have become part of Zen's consensus about the valid chain with most proof of work.

9.1 Application: Token sale

Bitcoin integration *makes it possible to sell tokens for bitcoins*. The sale contract checks that the purchase is in a Bitcoin block, creates some tokens, and sends them to a Zen address. The buyer's address can be encoded into the Bitcoin transaction. No update to the Bitcoin protocol is necessary.

The user finds the sale, then clicks a link to open it in their Zen wallet. The wallet checks the terms, then shows the user a purchase address. Once the user sends bitcoins to that address, the Zen wallet automatically claims the tokens. In the case of a capped sale, the sale contract can hold collateral to insure buyers who try to participate after the sale ends. The Zen wallet understands these terms and displays them to prospective buyers.

Other ways to conduct a token sale

If the buyer and seller are both online, the seller can create a signed promise to honour transactions to a particular Bitcoin address. The buyer can then purchase tokens *before* downloading the Zen wallet—the user experience is the same as for any other Bitcoin transaction. The buyer can download the Zen wallet later, open the promise, and claim the tokens.

9.2 Application: Sidechain

There are several ways to move value between Bitcoin and Zen. Assets on Zen that represent Bitcoin value are called *BitZen* assets, and are treated specially by the Zen wallet, which shows their risk profiles to their owners. BitZen can be used as surrogate bitcoins to back other assets or digital agreements that use Bitcoin as their currency.

One type of BitZen asset is created by a contract which holds other assets as collateral, such as the Zen native token or other BitZen assets. This contract issues BitZen when certain addresses are sent bitcoins, just as in the token sale application above. Unlike the token sale, anyone who owns that contract's BitZen can make an on-chain withdrawal request. This request starts a countdown in which the contract owner should send bitcoins to the requested address and provide the transaction to the contract. If the countdown ends, the contract uses some of the collateral to compensate the BitZen owner.

Another type of BitZen is created by a group of contract operators (a "federation")[8] using a single contract in common. Each operator is individually responsible for fulfilling withdrawal requests, and contributes collateral to the contract. In this case, however, if one operator fails to fulfill a withdrawal request, the BitZen owner can still turn to another operator instead.

9.3 Application: Simple cross-chain swaps

The Zen Protocol fully supports "atomic swaps" with other blockchains, where two parties use timelocked transactions to exchange assets on one chain for assets on another. In the case of Bitcoin, this is particularly simple: the "token sale" contract can operate with any asset it controls, allowing anyone holding a Zen asset to swap it for Bitcoins. A slightly more complex contract can support multiple sellers and buyers of different assets at different prices—which is only a short step away from the *token exchange* discussed below.

10 Using Zen

Zen supports many financial products. In this section, we discuss some examples of the assets and agreements possible on Zen, and how the Protocol makes it easier to discover and use them.

[8]More than one federation can exist. Each will issue its own BitZen asset.

10.1 A call option using a Merklized Oracle

A *call option* is a derivative which gives the holder to buy an asset at a set price. For example, a call option on stock X, with a set price (or *strike price*) of 160 dollars, gives the holder to buy one X stock for 160 dollars. The details of when the holder can use (or *exercise*) the option are different for different kinds of call option.

We discuss a variant of the call option, which doesn't give the holder the stock itself, but the notional value of the option. So if the stock is worth 180 dollars, and the holder exercises the option, the option pays 20 dollars "for free". As we mentioned in 7.3, contracts can hold collateral to fund these payouts.

There are three key requirements for our call option contract:

1. a source of collateral,
2. a data-feed to tell the contract the price of the stock, and
3. an interface to explain what the option does to the user.

10.1.1 Collateral

The collateral for the option comes from the creator of the contract. This creator wants to make profit from the other side of the option, in the case that the stock *doesn't* rise in value. The creator sends funds to the contract, along with an authenticated message telling the contract to use the funds as collateral. The contract has a rule which stops it creating options unless it has "enough" collateral.

10.1.2 Data-feed: the Merklized oracle

The contract needs to know the current price of the stock in order to calculate how much to pay the option holder, which requires a neutral, independent source for that data. This source, the *oracle*, continually creates *commitments* to the stock's price (as well as the price of many other stocks and assets), in a very efficient form: a Merkle tree. A Merkle tree is the same data structure Bitcoin uses for committing to transactions inside a block: it produces a single, 32-byte hash. When the option holder wants to exercise the option, she pays the oracle for an *audit path*—a proof that the price of the stock is covered by the commitment. This is an efficient way to use oracles: only the data that is actually used gets put into the blockchain, and all data is paid for by the one who uses it. One oracle commitment can cover many, many different options and asset types.

If the buyer and seller of the option don't want to rely on a single oracle, it's simple to use a contract which takes different oracles and listens to the majority result.

10.1.3 Interface: discovering and viewing an asset

The call option is of limited use unless potential buyers can find it, view it, and understand what it does. The contract writer makes this possible by writing a *proof* about what the contract does. The Zen wallet is preloaded with a definition of what call options do: when it sees the proof, it verifies it, then displays a *verified call option contract*, with information about the underlying stock, the collateral, and the oracles. The same information is available on the call options themselves. Users can download new asset type definitions to see new verified asset types.

Note that this doesn't mean matching the contract itself against a template: the contract author shows what the contract *does*, not how it does it. This lets you use a single contract to hold all your collateral, using it to sell options on multiple different stocks, for instance—without dividing your collateral. The user sees all the options in the Zen wallet as verified and available for purchase.

10.2 A token exchange

Even a small exchange executes thousands of trades per day.[9] This makes completely on-chain markets far too inefficient. Because blocks are mined at random intervals, and trades only become final when they appear in the blockchain, on-chain markets also introduce an unavoidable, random delay between the moment a trade becomes public and the moment it becomes finalized. *Front-runners* can observe these trades and take advantage of them, by trying to get their trades mined first. Front-runners are bad for markets: they neither offer liquidity nor provide price information (Harris, 2003, pp. 245&257). We discuss a simple token exchange that uses a central coordinator to keep most transactions off-chain, but that can't steal deposits.

Users open an account with the exchange by marking their funds with their own public keys, and sending them to the exchange's contract. They then establish state channels (Coleman, 2015) with the exchange. Each user can update the state channel with bids and asks: the exchange is responsible for executing trades by matching bids and asks. All the state channels share in common a Merkle root of all recent transactions: the exchange sends proofs to each user that their trades are present in the Merkle root. The exchange cannot steal users' assets: all withdrawals are made by sending a message to the contract proving that the state channel was closed. In case of a dispute, both the exchange and the users can prove to the contract that they acted correctly: if any party tries to use an old version of the state channel, a grace period lets others submit a fraud proof.

In this model, users should demand that the exchange hold some amount

[9]In a mature economy this is a significant underestimate: Bitfinex executes 4000 trades *per second*, while the NASDAQ executes approximately 1 MM trades per second.

of collateral. This collateral is not strictly necessary to protect users. Rather, it acts as a sort of buffer, making it easier for users to unwind trades quickly, and making it possible for them to maintain security while monitoring the exchange less frequently.

References

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4574-3. doi: 10.1145/2993600.2993611. URL http://doi.acm.org/10.1145/2993600.2993611.

Jeff Coleman. State channels, Nov 2015. URL http://www.jeffcoleman.ca/state-channels/.

Larry Harris. *Trading and exchanges.* Oxford University Press, 2003.

Johnson Lau. Merkelized abstract syntax tree, Apr 2016. URL https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki.

Adam Perlow and Nathan Cook. A proportionate response: Multi-hash mining, Mar 2017. URL https://www.zenprotocol.com/proportionateresponse.pdf.

Paul Sztorc. Drivechain: Enabling bitcoin sidechains, 2015. URL http://www.drivechain.info/literature/index.html.

Pieter Wuille. Base32 address format for native v0-16 witness outputs, Mar 2016. URL https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki#Bech32.