ZEN: Technical notes on a financial engine

Nathan Cook

July 18th 2017

Summary

We describe the Zen Protocol, the basis for a decentralized financial platform. This paper assumes more technical knowledge than the accompanying white paper, and may be read as a rough specification of the Protocol.

Contents

1 Introduction

Zen is a public blockchain, secured by proof of work, with powerful smart contracts and a practically unlimited number of assets.

Zen uses an “unspent transaction output” (UTXO) model, or a *coin-oriented* model. This means that instead of focusing on accounts and rules for accessing them, Zen knows about coins and the rules for spending them. This is the same design choice as for Bitcoin, preventing several problems affecting account-oriented blockchains.

Bitcoin locks coins with *Script*, a simple stack-based language without recursion. In contrast, in the Zen protocol, output locks don’t contain opcodes. Instead, output locks can either be unlocked with a public key signature, or by a named **contract**. Contracts decide whether to unlock an output by processing information present in the blockchain, including data attached to the output itself.

Zen contracts are black boxes: once made, they act as pure functions which generate transactions. They are not timed by Zen nodes, and how they generate their output is not tracked by the Zen protocol consensus. Zen limits the resource consumption of contracts by requiring that they prove what resources they consume *once*, at the moment they enter consensus.

Contracts remain active and able to affect the blockchain for a limited amount of time. They pay miners for each block they remain in the **active contract set**. Contracts are written in F*, then compiled by nodes when they activate. A contract can be extended by an additional payment, labeled with its identifying hash, but contracts which deactivate can only be reactivated by providing their source code once more.

The Zen protocol directly incorporates the state of the Bitcoin blockchain by reference. Each Zen block may commit to one or many Bitcoin blocks by including their block headers. Zen nodes should validate the Bitcoin blockchain as well as the Zen blockchain—simply validating the Bitcoin proof-of-work is not advisable to a miner, but may be perfectly acceptable for a non-mining node or a thin client. Zen contracts can take serialized Bitcoin transactions, prove them to be present in certain Bitcoin blocks, and alter their behaviour depending on the Bitcoin blockchain.

The Zen protocol supports many different types of asset. Just as each contract is identified by its hash, the assets it creates are likewise identified by that same hash, plus an index.[1] A contract with hash H may create, or destroy, as much as it pleases of any asset whose identifier begins with H.

The asset with identifier (0,0) is the *Zen token*, which is generated by mining and is used to activate contracts.

2 Hashing

Our cryptographically secure hashing algorithm is **SHA-3**, with 256-bit output. SHA-3 is not susceptible to length-extension attacks, an occasional source of unpleasant vulnerabilities in blockchain protocols. Additionally, we use **tagged hashing** for all data types, by adding a short type-specific prefix to each serialized object. This makes it unfeasible to create any two distinct objects, or lists of objects (including empty lists) sharing the same hash [Todd, 2016]. The tags are:

Transaction	tx
OutputLock	ol
Spend	sp
Outpoint	outpoint
Output	output
Contract	contract
BlockHeader	block

Tagged hashing is particularly useful for the construction of "efficient sparse merkle trees", used for commitments to the active contract set, as described below.

3 Serialization

Almost every Zen data structure is serialized according to the MessagePack format. A widely used, extensible, efficient binary serialization format, Mes-

[1]Each contract may create up to 2^{256} types of asset.

sagePack does suffer from a lack of canonicity—more than one serialization can map to the same deserialized data structure. We evade this issue by requiring that any packed Zen structure match itself when round-tripped through deserialization and reserialization, thus establishing a canonical serialization. This requirement can be checked very easily by performing the round-trip, but more quickly with a smarter deserializer. The 1.0 release of Zen will use a fast, formally verified implementation of MessagePack that enforces a canonical form for each data structure.

4 Digital signature scheme

We use ed25519, as implemented by libsodium. This provides 32-byte public keys and 64-byte signatures. Future releases may include other signature schemes via soft-fork.[2] All cryptographic primitives used in Zen can be used by Zen contracts.

5 Blocks

Let us now look at how the Zen blockchain is produced.

The Zen blockchain begins with a **genesis block** to which very minimal validation is applied. Any valid blockchain must be rooted at the genesis block. Every non-genesis block must conform to rules about:

1. Structural validity
2. Proof of work
3. Merkle root commitments
4. Transaction validity
5. Other chain state validity

We deal with each of these in turn.

5.1 Structural validity

This simply refers to the requirement that each block be validly formatted. Here is the schematic block structure.

[2]The principle candidates are Schnorr signatures for multisignature aggregation and other improvements, and, over the medium term, a post-quantum scheme based on supersingular isogeny.

⟨block⟩	⊨	⟨block-header⟩ ⟨transaction-list⟩
⟨block-header⟩	⊨	⟨block-version⟩ ⟨parent⟩ ⟨block-number⟩ ⟨commitments⟩ ⟨timestamp⟩ ⟨target⟩ ⟨nonce⟩
⟨block-version⟩	⊨	*block version: uint32*
⟨parent⟩	⊨	*256-bit hash of parent block*
⟨block-number⟩	⊨	*number of blocks from genesis block*
⟨commitments⟩	⊨	⟨transaction-merkle-root⟩ ⟨witness-merkle-root⟩ ⟨contract-merkle-root⟩ ⟨extra-data⟩
⟨*-merkle-root⟩	⊨	*256-bit merkle root*
⟨extra-data⟩	⊨	*extra commitments, including Bitcoin block headers*[2]
⟨timestamp⟩	⊨	*100-ns intervals since 0001-01-01-00:00UTC. int64*
⟨target⟩	⊨	*compressed target: uint32*
⟨nonce⟩	⊨	*80-bit field, to be varied by miners when hashing*
⟨transaction-list⟩	⊨	⟨coinbase-transaction⟩ ⟨other-transactions⟩
⟨coinbase-transaction⟩	⊨	*first transaction in block*
⟨other-transactions⟩	⊨	⟨transaction⟩ ⟨other-transactions⟩ | ϵ

5.2 Proof of work

Zen's **PoW** puzzle is currently set as SHA-3. Before version 1.0 is released, this will change to the multi-hash scheme described in section 8.2. The puzzle's difficulty is encoded as a *compressed target*, in a similar manner to Bitcoin: the upper byte of the compressed target is an exponent, the lower three bytes a significand. The decompression algorithm is:

Algorithm 1 Decompress target to bigint

```
1: procedure DecompressTarget
2:     $upperByte \leftarrow target_{compressed} \parallel \texttt{0xff000000}$
3:     $exp \leftarrow (upperByte \gg 24) - 24$
4:     $significand \leftarrow (target_{compressed} \parallel \texttt{0x00ffffff}) + 1$
5:     return $significand * 2^{exp} - 1$
6: end procedure
```

The corresponding compression algorithm takes a bigint and returns an unsigned 32-bit integer.

[2]Specification of commitments is not finalized. This section will be enhanced as the protocol is refined.

Algorithm 2 Compress target to uint32

```
Require: $1 \leq target < 2^{256}$
procedure CompressTarget
    $x \leftarrow target + 1$
    $t \leftarrow 0$
    while $2^t \leq target$ do
        $t \leftarrow t + 1$
    end while
    if $t < 24$ then
        $sig \leftarrow x$
        $exp \leftarrow 24$
    else
        $sig \leftarrow x \gg (t - 24)$
        $exp \leftarrow t$
    end if
    $sig \leftarrow sig - 1$
    return $(exp \ll 24) + sig$
end procedure
```

The **network difficulty** is defined as 2^{256} divided by the uncompressed target—the greater the difficulty, the harder it is to find a block, as one would expect. Each block commits to the network difficulty by the compressed target in its header. A valid block commits to the correct network difficulty as calculated by the update algorithm given in section 8.1.

5.3 Merkle root commitments

Aside from commitments to *difficulty*, to *block number*, and to *the state of the Bitcoin blockchain*, each block also commits to:

1. a list of transactions
2. a list of witnesses—that is, data used in validating transactions
3. the Active Contract Set (**ACS**).

Each commitment is via a *merkle tree*, a data structure used in Bitcoin (and most other blockchains) to commit to a list of transactions. At the time of writing, Bitcoin is in the middle of a potential activation of the *Segregated Witness* (SegWit) soft-fork, which moves a transaction's witnesses (which are normally signatures or collections of signatures) to an area outside of the transaction itself—enabling improvements to the efficiency and immutability of transactions. A Bitcoin block must still commit to the witnesses in order to permit validation of their transactions. For backwards-compatibility, Bitcoin will place this commitment in the coinbase transaction. In Zen, all transactions put signatures in dedicated witness fields, and the witness commitment appears in the block header.

For an explanation of the use of merkle trees to calculate a commitment to a list of data, we recommend Becker [2008]. Zen's merkle tree fixes a small flaw in Bitcoin's implementation: rather than duplicating the last item of an odd list—if one should appear at some iteration of the merkle root algorithm—we complete any list of items to an exponent of 2, adding *empty hashes*. Note that these are *tagged hashes*, so the hash of an empty witness is different from that of an empty transaction.

5.3.1 ACS commitments with Efficient Sparse Merkle Trees

The remaining merkle root commitment in the initial release of Zen is to the *active contract set*. The ACS at any point in time is the set of contracts capable of generating autotransactions. This set changes *within* each block and *between* blocks as contracts activate and deactivate. The rules for contract activation, extension and deactivation are part of the Zen consensus, but we commit to the contract set in each block for a several reasons—to make lightweight wallets possible, to make a consensus fork immediately detectable, and to make block validation faster.

Each block commits to the active contract set *after all transactions in the block, and then after all inactivating contracts are removed from the set.* In addition to committing to the contracts themselves, we commit—in the same merkle root—to a small amount of metadata for each contract, namely the block number after which it will expire. We explain further in section 6.

The conventional merkle tree data structure is inefficient for committing to the ACS, in which many items persist from one block to the next. Moreover, the ACS may grow quite large, making block validation slower as the blockchain is extended. Ethereum has a similar problem (exacerbated due to contracts never expiring) and introduced the *Patricia tree*, a modification of the radix tree, as a solution. However, Patricia trees, while efficient, lack (along with conventional merkle trees, in fact) an intriguing property—the ability for a node to efficiently generate a short proof of *non-membership*. This is a very useful property for a lightweight client, which can now validate autotransactions without itself keeping track of the ACS. It is even possible to efficiently incorporate validation of a proof of (non-)membership into a contract itself, which expands the utility of contracts. Recent work [Dahlberg et al., 2016] introduces a data structure called the **efficient sparse merkle tree**, which provides fast generation of proofs of membership and non-membership, along with fast insertions and deletions. In addition, the authors prove that the security of each function of their data structure reduces to the security of the underlying hash algorithm.

Here is a brief description of the operation of the efficient sparse merkle tree (ESMT), and how Zen uses this structure to commit to the ACS. Each contract has a unique tagged hash. Much as in a (merklized) radix tree, rather than storing contracts at the leaves of a tree, we use the hash of a contract as a pointer to a leaf, at which we attach an empty hash if there is no active contract, and the hash of some non-empty value if there is a contract. In the case of the ESMT, we use a *complete binary tree* of height 256. Such a tree has 2^{256} leaves and so must be stored as a sparse data structure. We pre-compute *empty digests*, i.e. the merkle roots of empty trees of height 0, 1, 2, etc., up to 255. Using these, and some cached non-empty digests, we can efficiently compute all required data for modifying the ESMT and generating *audit paths*—the proofs of membership and non-membership.

As mentioned above, we commit to some meta-data for each active contract, making our active contract *set* more properly a *key-value store*. Doing so transforms the ESMT into a *persistent authenticated dictionary* (PAD). The extension to a PAD is suggested in Dahlberg et al. and does not affect the proof of security reduction.

5.4 Transaction validity

A block is invalid if any of its transactions are. Transaction validation is dealt with in section 7.

5.5 Other chain state validity

In addition to the validity rules given above, each block may contain Bitcoin block headers. *These headers are validated according to the full Bitcoin consensus rules*, meaning that their transactions must also be valid! We forgo the re-implementation of an entire Bitcoin node to achieve this, and instead provide an RPC interface to talk to an existing Bitcoin client and obtain valid block headers. The consensus Bitcoin chain, as far as *Zen* consensus is concerned, is the valid Bitcoin chain committed to in Zen block headers with the most proof of work. Miners are rewarded for including valid block headers by being able to claim more Zen tokens in the coinbase transaction.

6 Contracts

When talking about a blockchain, there are good contracts and bad contracts. Good contracts run without inconveniencing any node or miner, pay for their keep, and use no more resources than they need to. Bad contracts break these rules.

The Zen protocol requires contracts to prove that they are good—that is, well-behaved citizens of the Zen ecosystem. This is a key distinction from protocols which allow contracts to be stopped in the middle of operation.

We discuss contracts in stages. First, we discuss the contract lifecycle—how they enter and leave the active contract set. Secondly, we talk about how contracts operate to put transactions into the blockchain. Lastly, we tackle how contracts are created, and how they prove that they don't consume excessive resources.

6.1 The contract lifecycle

Zen tracks contracts in the *active contract set*(ACS), the set of those contracts which can themselves affect the Zen consensus. A contract enters the ACS when a transaction includes its source code and pays a **contract sacrifice**. That contract sacrifice then pays miners for the time that the contract remains active. The contract becomes inactive again when the contract sacrifice is depleted.

6.1.1 Activation

A transaction activates a contract by providing its source code, proving its good behaviour, and paying a contract sacrifice to keep it alive for some number of blocks. If the transaction successfully proves that a contract has the necessary properties to enter the Zen consensus, the sacrifice is *restricted*, meaning that miners can only claim a portion of the sacrifice each block. The size of the sacrifice and the rate at which it can be claimed depends on the size of the source code and how much computation is needed to verify the proof.

If a contract either fails to validate, or has a higher version number than is currently defined, then the contract sacrifice is unrestricted, and can be claimed by any miner. That is to say: the active contract set only contains validated contracts with defined behaviour.

The contract field of a transaction is parsed as follows:

$$\begin{aligned} \langle\text{contract}\rangle &\models \langle\text{code}\rangle\ \langle\text{bounds}\rangle\ \langle\text{hint}\rangle \mid \langle\text{high-v-contract}\rangle \\ \langle\text{high-v-contract}\rangle &\models \langle\text{contract-version}\rangle\ \langle\text{contract-data}\rangle \end{aligned}$$

Here `code` is the source code of the contract-function, `bounds` is a commitment to the amount of computation needed to validate the contract, and `hint` is an additional data field containing "hints", which reduce the computation needed to validate the contract.

The contract sacrifice required to activate a contract for n blocks, including the current block, is $\alpha * \mathit{bounds} + \beta * (n-1) * \mathit{size}$, where α and β are constants. This sacrifice is paid in the Zen token. Contracts must be activated for at least *two* blocks, i.e. the activation block and the following block. This enforces the property that each change to the ACS is recorded in at least one block.

6.1.2 Extension

Contracts may be extended by putting their identifier in the data field of a transaction's contract sacrifice lock. Any such lock has its sacrifice counted towards the referenced contract, rather than the contract, if any, in the transaction's contract field. Only active contracts can be extended. The number of additional blocks for which an extended contract remains active is calculated based solely on the size of the contract's source code in bytes. It costs $\beta * n * \mathit{size}$ to extend a contract for n blocks, where β is the same constant as above. This makes the marginal cost to keep a given contract in the ACS equal, whether extending or activating.

6.1.3 Deactivation—and reactivation

The ACS tracks the last block in which each contract is active. If not extended, at the end of this block, the contract becomes inactive, leaving the ACS. Once inactive, nodes are no longer required to track the contract in memory or even on disk, and reactivation requires exactly the same conditions as the activation of a fresh contract.

6.2 Contract execution

Contracts only affect consensus via transactions. Zen nodes don't care where these transactions come from—only that the contracts *could* have generated them. Some transactions may identify themselves as created by several contracts working together, or by contracts and users—we deal with these cases in section 7.5. What about the simplest case, when a transaction is generated by only one contract?

Suppose a node sees a transaction, T whose inputs are all locked to the same contract, C. How should it validate this transaction? Simple: try to reproduce the same transaction by running the contract. If C is inactive, then it can't generate T, and won't accept T in a block. But if the contract is active, then we treat it as a pure function accepting some variables and returning a **reduced transaction**—a transaction with no witnesses. When validating an autotransaction[3], a node:

1. finds the arguments for the contract function
2. passes them to the contract function
3. checks that the result equals the transaction with witnesses stripped.

The arguments for the contract function comprise the context in which it runs, together with a message specific to the transaction. The message can be either in a witness, or in the lock of an input spent by the transaction.

[3]A transaction generated by a contract.

6.2.1 Witness data as argument to contract function

If a node sees an autotransaction with a non-empty first witness, it uses that witness as a message to the contract function. This form of autotransaction can only be created by a user (who may be using an automated—off-chain—process to create it).

6.2.2 Lock data as argument to contract function

If a node sees an autotransaction whose first witness is empty, it looks at the first input spent by that transaction, takes the data field from its lock (which will be a contract lock), and passes it to the contract as the message. This form of autotransaction can be created by users and by other contracts[4].

6.2.3 Errors and time-outs

What happens if, instead of returning a reduced transaction, the contract-function raises an exception, or runs forever? The answer is that *it never does.* In order to activate a contract—to let it generate transactions—the activating transaction must prove both that it never throws an error, and that it always terminates. In fact, it must prove *how long* it takes to terminate—see subsection 6.4.

6.2.4 Contract context

In addition to a per-transaction message, contracts also know about some of the current blockchain state. Contracts have access to their own hash, to all UTXOs, to the headers of the last 500 blocks, and to the hashes of all blocks in the blockchain.

The basic type signature of a contract function is

```
(message:array byte
* contractId:array byte
* utxoMap:(outpoint -> option output)
* list blockHeader
* blockHashes:list (array byte))
->
( list outpoint
* list output
* option contract)
```

Note the `Contract option` field: autotransactions can activate new contracts.

[4]Miners or other nodes can opportunistically try to use the contract lock data they see as messages to contract functions. Future contract and transaction versions may support transaction outputs which require another autotransaction to consume them immediately, allowing for guaranteed chained contract execution.

6.3 Writing a contract

Contracts are written in a dialect of the dependently typed functional language, F* (pronounced F-star), with a custom standard library. Programming in F* is quite like programming in ML, OCaml, or F#. The dialect excludes unsafe expressions which could break Zen's security guarantees. Additionally, many library functions specific to contract programming are included: SHA-3, EdDSA, etc.

6.4 Complexity bounds: proof and verification

Any program written for the Zen blockchain must have a specific type. The Zen dialect of F* is capable of expressing resource use in the type of a program, and proving that these types accurately describe the behaviour of the program. The type signature of an F* contract is the basic type signature given at at 6.2.4, annotated with a computation cost and some information about the arguments. For example:

```
(message:(n:nat & array byte n)
* contractId:array byte 32
* utxoMap:(outpoint -> option output)
* list blockHeader
* blockHashes:list (array byte 32))
-> cost
( list outpoint
* list output
* option contract) (20 + 3 × n)
```

In the above example, the type of `message`, a byte array, is refined to carry its length `n`. This number `n` is then used as part of the function's return type, specifying it to have a cost of $20 + 3 \times$ `n`. This construction is only possible in a dependently typed language like F*.

Complexity is represented by a notional count of the work to perform each evaluation or reduction.[5] The version 1 complexity model attempts to produce bounds no more than a factor of about 2 away from the empirical cost of running the contract. Later contract versions will likely attempt to assign prices to each operation via the operation of a market, but the mechanism design for this is considerably more complex. In any case, Bitcoin, which has fixed, hard limits on "opcode count", shows that a roughly accurate system can be sufficient.

Most of the time, developers can just specify a cost for their contracts and allow F* to prove that the cost is correct. This is possible because F*

[5]The version 1 cost model is represented by a generalized monad similar to that described in Danielsson, with costs corresponding to the instrumental semantics given in Rosendahl. Future contract versions will use the generalized monad given in McCarthy et al.

has sufficient automated proving abilities to infer bounds on most contracts. A particularly unusual or complicated contract may require the developer to help F* prove a complexity bound. This is done by writing *lemmas*, or *annotations*, on intermediate expressions. The prover uses these proofs to guide its search.

6.5 Under the hood

When a contract is activated, the business of validating the various necessary proofs is delegated to a separate F* process. This process runs Z3, an automated theorem prover, to prove the contract's assertions. In *publish* mode, the Zen client instructs the Z3 solver to run with configurable resource limits, creating hints as it goes. These hints tell the Z3 solver the most efficient way to construct a proof, without following any of the dead ends in the initial search.

The client then uses the hints to calculate the required Zen sacrifice to activate the contract and creates the activating transaction.

The output of the F* process is F# source code. The node takes this code and compiles it on the fly to Common Language Runtime code, then caches the resulting code object. The same process is applied to the bounds themselves, returning a simpler program guaranteed to calculate a complexity bound within a few hundred CPU cycles.

When an autogenerated transaction is received, miners run the complexity function to check the upper bound, then decide whether or not to attempt validation and include the transaction in the next block. Transactions are verified using the compiled CLR code objects. This makes contract code approximately as fast as other compiled binaries.

6.6 Protocol upgrades

High version contracts are considered to be capable of doing anything and always to be active. Assets locked to a high version contract can be taken by anyone on the network. Any contract sacrifice with a high version lock can similarly be immediately claimed by any miner. This theoretically enables any new contract language to be implemented by soft-fork—in practice, it is likely to be used initially to tweak parameters and add library functions. A key improvement that may be incorporated into the 1.0 release is the ability for a contract to specify how much of the Bitcoin chain state or Zen chain state it wants to have access to, paying a different sacrifice accordingly.

6.7 Economic limits

To prevent DOS attacks, centralization, etc. a limit is imposed on the total amount of computation in one block. For autotransactions, the amount of computation is given by the cost bound function, rather than the actual amount of computation taken by that specific autotransaction validation. This makes it possible for miners to select transactions for inclusion in a block based on fees and computation before running the contracts, which improves efficiency. User-generated transactions are counted too, at a reduced rate: each PKLock on each input is counted as having the same computational complexity as one hash and one signature validation operation. High version transactions are set to have zero complexity. This enables the computation limit effectively to be raised via soft fork, if necessary.

The per-block computation limit is accompanied by a soft block size limit which halves mining reward for every megabyte above 2MB. Note that this limit will become less economically binding as the maximum mining reward decreases.

7 Transactions

Zen transactions do several things. They move assets, as in Bitcoin. They may activate and extend contracts. They may put raw data into the Zen consensus. Some transactions may create tokens.

7.1 Structural validity

As with blocks in the previous section, we give a schema for the structure of a transaction.

$$
\begin{array}{rcl}
\langle\text{transaction}\rangle & \models & \langle\text{transaction-version}\rangle\ \langle\text{input-list}\rangle\ \langle\text{witness-list}\rangle \\
 & & \langle\text{output-list}\rangle\ \langle\text{optional-contract}\rangle \\
\langle\text{transaction-version}\rangle & \models & \textit{transaction version: uint32} \\
\langle\text{input-list}\rangle & \models & \langle\text{outpoint}\rangle\ \langle\text{input-list}\rangle \ \mid\ \epsilon \\
\langle\text{witness-list}\rangle & \models & \langle\text{witness}\rangle\ \langle\text{witness-list}\rangle \ \mid\ \epsilon \\
\langle\text{output-list}\rangle & \models & \langle\text{output}\rangle\ \langle\text{output-list}\rangle \ \mid\ \epsilon \\
\langle\text{optional-contract}\rangle & \models & \langle\text{contract}\rangle \ \mid\ \epsilon \\
\langle\text{outpoint}\rangle & \models & \langle\text{tx-hash}\rangle\ \langle\text{index}\rangle \\
\langle\text{tx-hash}\rangle & \models & \textit{256-bit transaction hash} \\
\langle\text{index}\rangle & \models & \textit{index of output in given tx-hash: uint32} \\
\langle\text{witness}\rangle & \models & \textit{witness data: byte[]} \\
\langle\text{output}\rangle & \models & \langle\text{output-lock}\rangle\ \langle\text{spend}\rangle \\
\langle\text{output-lock}\rangle & \models & \langle\text{fee-lock}\rangle \ \mid\ \langle\text{contract-sacrifice-lock}\rangle \ \mid \\
 & & \langle\text{pk-lock}\rangle \ \mid\ \langle\text{contract-lock}\rangle \ \mid\ \langle\text{high-v-lock}\rangle \\
\langle\text{spend}\rangle & \models & \langle\text{asset}\rangle\ \langle\text{amount}\rangle \\
\langle\text{asset}\rangle & \models & \langle\text{contract-hash}\rangle\ \langle\text{asset-index}\rangle \\
\langle\text{asset-index}\rangle & \models & \textit{256-bit asset index} \\
\langle\text{amount}\rangle & \models & \textit{quantity of asset locked: uint64} \\
\langle\text{fee-lock}\rangle & \models & \text{FeeLock}\ \langle\text{lock-core}\rangle \\
\langle\text{contract-sacrifice-lock}\rangle & \models & \text{ContractSacrificeLock}\ \langle\text{lock-core}\rangle \\
\langle\text{pk-lock}\rangle & \models & \text{PKlock}\ \langle\text{pk-hash}\rangle \\
\langle\text{contract-lock}\rangle & \models & \text{ContractLock}\ \langle\text{contract-hash}\rangle\ \langle\text{contract-data}\rangle \\
\langle\text{high-v-lock}\rangle & \models & \text{HighVLock}\ \langle\text{lock-core}\rangle\ \langle\text{type-code}\rangle \\
\langle\text{lock-core}\rangle & \models & \langle\text{lock-version}\rangle\ \langle\text{lock-data}\rangle \\
\langle\text{lock-version}\rangle & \models & \textit{lock version: uint32} \\
\langle\text{lock-data}\rangle & \models & \textit{lock data: byte[]} \\
\langle\text{pk-hash}\rangle & \models & \textit{256-bit public key hash} \\
\langle\text{contract-hash}\rangle & \models & \textit{256-bit contract hash} \\
\langle\text{contract-data}\rangle & \models & \textit{contract-directed data: byte[]} \\
\langle\text{type-code}\rangle & \models & \textit{lock type identifier: uint32}
\end{array}
$$

We see that all transactions have *inputs*, *witnesses* and *outputs*, and may have one *contract*. Contract structure is given in section 6. An additional structural rule not given above is that the length of the input list and the witness list must be equal.

Inputs are references to existing unspent transaction outputs (UTXOs), as they are in Bitcoin. Each such reference consists of a transaction identifier and the index of the output within that transaction.

Witnesses are raw data. Their interpretation depends on the lock.

Outputs contain **spends**, which specify *what* is being put into the UTXO set, and **output locks**, which give the conditions to *use* the spend in question.

The contents of the **contract** field do not affect the validity of a transaction, although they do affect, of course, what happens to the contract. This provides forward-compatibility by allowing soft-forks to define new contract versions. This document describes version 1 contracts.

Output locks are all serialized with a *version*, a *type code*, and *lock data*, but we impose additional structural validity rules on version 1 locks, corresponding to the definitions given above—for instance, a *PKLock* is defined as any lock with version 1 and type code 3, and is then required to have *lock data* of length exactly equal to 32 bytes, which is the length of a pubkey hash.

7.2 Coinbase transaction validity

After *structural* rules are checked, a coinbase transaction is subject to more checks. Firstly, it must contain no inputs (and therefore no witnesses). Secondly, the spends of its outputs are checked. A coinbase transaction may claim:

1. any fees in its block, no matter what asset they are paid in
2. a set miner reward, paid in Zen tokens
3. any unrestricted contract sacrifices
4. any restricted contract sacrifices to which it is entitled.

The meaning of contract sacrifices is given in section 6. Fees are not claimed by spending the outputs with a fee lock, but rather in each block the total amount of each asset locked with a fee lock is calculated and added to the per-asset maximum on what the coinbase transaction may generate. This is a similar set-up to that of Bitcoin, except that all fees must be explicitly allocated in each non-coinbase transaction.

The outputs of coinbase transactions may not be spent by any transaction whatsoever for 100 blocks.

7.3 User-generated transaction validity

Any transaction whose inputs have no contract locks is counted as user-generated. Such transactions may neither create *nor destroy* any assets: any left-over value in any spend must be allocated to a fee-locked output or locked to a change address. In practice, this means user-generated transactions are restricted to spending PKLock inputs.[3] These transactions are validated similarly to Bitcoin segwit transactions:

1. Each witness is parsed as a pubkey, a signature and a sighash type.
2. According to the sighash type, a subset of the transaction is serialized and cached. In every case, the witnesses are stripped.
3. Each witness is checked against the corresponding input for matching pubkey.
4. Each witnesses signature is checked against the serialized transaction and the provided pubkey.

Sighash types are an idea from Bitcoin, where users specify which parts of transactions they sign. For instance, a user can indicate that the signature covers only one input and one output, without caring about other inputs and outputs. Zen's sighash types are the same as those in Bitcoin, encoded the same way: `SIGHASH_ALL` is 0x1, `SIGHASH_NONE` is 0x2, `SIGHASH_SINGLE` is 0x3, and `SIGHASH_ANYONECANPAY` is 0x80.

Caching the transaction avoids $O(n^2)$ complexity in validating transactions with many inputs.

7.4 Autotransaction validity

Any transaction with at least one contract-locked input is validated as an autotransaction, or contract-generated transaction. Autotransactions obey two rules: they create only the assets with identifier equal to the identifier of their contract(s), and their contract(s) are capable of generating them, when given some data as input. How this occurs is a main subject of section 6.2.

7.5 Co-operative transaction validation

Contracts and users can co-operatively create or sign a transaction. In this case, each signature must sign the necessary parts of the transaction, while each contract must generate the entire transaction when run with the specified arguments.

7.6 Validation

The algorithm for validating a transaction is as follows:

[3] Multi-signature locks will be added at a later stage of development, but much the same functionality is available immediately by using a simple contract.

Algorithm 3 Validate transaction, Part 1

```
procedure ValidateTransaction
    n ← length(transaction.inputs)
    digests ← Map<Transaction,hash>.init
    executed ← Set<hash>.init
    for $i = 0$ to $n - 1$ do
        current_input ← transaction.inputs.[i]
        lock ← current_input.lock
        witness ← current_input.witness
        if lock unspendable then
            return false
        else if lock.version > 1 then
            continue
        else if lock matches PKLock then
            reduced_tx ← reduce(transaction,i,witness.sighash)
            if reduced_tx in digests then
                digest ← digests.[reduced_tx]
            else
                digest ← hash(reduced_tx)
                digests.[reduced_tx] ← digest
            end if
            if validate_pk(digest, lock.pkhash, witness) then
                continue
            else
                return false
            end if
```

(Algorithm continued on following page.)

Input locks are validated using the corresponding witness. Inputs locked with public key hashes are verified independently of each other. Their corresponding witnesses carry sighashes indicating which inputs and outputs they sign. An input locked with a contract lock is validated by running the contract on data stored either in the witness or the lock itself, as described in 6.2. However, if the contract has already been used to validate an input, it runs again only if the witness is not empty.

Provided that a contract validates at least one of the inputs of a transaction, that transaction may create or destroy any amount of that contract's assets.

Algorithm 4 Part 2

```
        else if lock matches ContractLock then
            contracthash ← lock.contracthash
            if not active(contracthash) then
                return false
            else if contracthash in executed and witness = [] then
                continue
            else if witness = [] then
                msg ← lock.data
            else
                msg ← witness
            end if
            if exec(contracthash, msg) ≠ strip(transaction) then
                return false
            else
                add contracthash to executed
                continue
            end if
        else
            return false
        end if
    end for
    return true
end procedure
```

8 Chain state

Aside from transactions and contracts, there are two other large elements of chain state: mining difficulty and Bitcoin state.

8.1 Updating network difficulty

Network difficulty is adjusted each block using an exponential moving average of inter-block interval. The EMA block interval is calculated as:

$$\tau_n = \begin{cases} 0.95\tau_{n-1} + 0.05t_n, n < 100 \\ 0.999\tau_{n-1} + 0.001t_n, n \geq 100 \end{cases}$$

where τ_n is the estimator of inter-block interval for block n, and t_n is observed inter-block interval for block n. τ_0 is set to five minutes. We target a block time of five minutes, giving us a *target multiplier* of $\frac{\text{five minutes}}{\tau_n}$. We clamp this multiplier to the range $[0.95, 1.05]$ and multiply the previous block's network difficulty by it to calculate the current block's network difficulty.

8.2 Multi-hash mining

The Zen protocol will use the multi-hash mining algorithm described in Perlow and Cook [2017]. Briefly, any block can be mined using one of a number of hash algorithms. Holders of the Zen token gain the right to vote for what the ratio should be between those algorithms, and the difficulty of each hash algorithm adjusts to enforce the result of this vote.

In Zen, one vote is held in each 2000 block period. Votes are counted from the 500th until the 1500th block, and the ratio adjustment occurs at the start of the next period. Zen tokens do not grant voting rights until held for 1000 blocks without moving. Ratio adjustment targets the result of a vote gradually, with no vote able to raise or lower difficulty by more than 20% in any voting period. Contracts and users are equally able to vote. Voting via contracts allows, among other things, for a user to put Zen tokens into cold storage while still voting with them.

A vote is recorded on a special unspendable output with the identifier (contracthash=1,index=0). The output is given a contract lock with contracthash set to hash 0x1, with the vote itself, a list of natural numbers, serialized into the data field of the lock.

8.3 Bitcoin integration

The very tight integration of Bitcoin into the Zen protocol enables many new contract classes using Bitcoin as collateral against assets in Zen, and vice versa. It is very simple to write a contract that operates only if some particular coins move on the Bitcoin network, for example.

Zen's consensus on the Bitcoin blockchain is co-ordinated by commitments to Bitcoin blocks. Miners include Bitcoin block headers in their own Zen block headers in return for a reward. Any valid Bitcoin block header may be included in a Zen block, but the Zen consensus only acknowledges a change to the Bitcoin blockchain if the corresponding Bitcoin block is valid and on the chain with most proof of work. Zen does not commit to the *validity* of a Bitcoin block until its timestamp is more than 5 days in the past. This delay reduces the effect of Bitcoin chain reorganizations, and reduces the impact of Bitcoin block withholding attacks.

References

Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep.*, 2008.

Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, volume 43, pages 133–144. ACM, 2008.

Jay McCarthy, Burke Fetscher, Max S New, Daniel Feltey, and Robert Bruce Findler. A coq library for internal verification of running-times. *Science of Computer Programming*, 2017.

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

Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156. ACM, 1989.

Peter Todd. Code review: The consensus critical parts of segwit, Jun 2016. URL https://petertodd.org/2016/segwit-consensus-critical-code-review.