The Ergo Yellow Paper

Ergo Developers

This is an early draft of the yellow paper

Contents

Part I
Introduction

This document describes a blockchain protocol named Ergo Platform (or just Ergo) and a corresponding reference implementation. Ergo is a conservative Proof-of-Work blockchain with ultimate new contract language (called ErgoTree, but in most cases you need a higher-level language, for example, ErgoScript) and better support for lightweight and secure modes of operation.

Vision

State-Oriented Design

Part II
The Protocol

Notation and Preliminaries

A Note on Notation

Cryptographic Primitives

We use Blake2b hash function with 256 bits output.

Modes of Operation

Ergo (since the very first testing network Testnet0) is supporting multiple security models. In addition to full node mode, which is similar to Bitcoin fullnode, Ergo reference implementation supports Light-SPV, Light-Fullnode and Pruned-Fullnode modes.

Full-Node Mode

Like in Bitcoin, a full node is storing all the full blocks since genesis block. Full node checks proofs of work, linking structure correctness (parent block id, interlink elements), and all the transactions in all the blocks. A fullnode is storing all the full blocks forever. It is also holding full UTXO set to be able to validate an arbitrary transaction. The only optimization a fullnode is doing is that is skipping downloading and checking AD-transformation block part (see below in the "Light-Fullnode" section). For the full node regime, modifiers processing workflow is as follows:

1. Send ErgoSyncInfo message to connected peers.
2. Get response with INV message, containing ids of blocks, better than our best block.
3. Request headers for all ids from 2.
4. On receiving header:

```
if(history.apply(header).isSuccess) {
    if(!isInitialBootstrapping) Broadcast INV for this header
    Request transaction ids from this block
 } else {
    blacklist peer
 }
```

5. On receiving transaction ids from header:

```
transactionIdsForHeader.filter(txId => !MemPool.contains(txId)).foreach { txId =>
  request transaction with txId
 }
```

6. On receiving a transaction:

```
 if(Mempool.apply(transaction).isSuccess) {
  if(!isInitialBootstrapping) Broadcast INV for this transaction
  Mempool.getHeadersWithAllTransactions { BlockTransactions =>
     GOTO 7
  }
}
```

7. Now we have BlockTransactions: all transactions corresponding to some Header

```
if(History.apply(BlockTransactions) == Success(ProgressInfo)) {
  if(!isInitialBootstrapping) Broadcast INV for BlockTransactions
 /*We should notify our neighbours, that now we have all the transactions
 State apply modifiers (may be empty for block in a fork chain)
 and generate ADProofs for them.
 TODO requires different interface from scorex-core,
 because it should return ADProofs
 TODO when minimal state apply Progress info,
 it may also create UTXOSnapshot
 (e.g. every 30000 blocks like in Ethereum).
 This UTXOSnapshot should be required for mining by Rollerchain*/
 if(State().apply(ProgressInfo) == Success((newState, ADProofs))) {
   if("mode"="full" || "mode"=="pruned-full") ADProofs.foreach ( ADProof => History.apply(ADPro
   if("mode"=="pruned-full" || "mode"=="light-full") drop BlockTransactions and ADProofs older
 } else {
   //Drop Header from history, because it's transaction sequence is not valid
```

```
      History.drop(BlockTransactions.headerId)
    }
  } else {
    blacklist peer who sent header
  }
```

Pruned Full-Node Mode

This mode is similar to fast-sync in Geth or Grothendieck, warp-mode in Parity (all the three are Ethereum protocol clients), but makes more aggressive optimizations. In particular, a pruned-fullnode is not downloading and storing full blocks not residing in a target blockchain suffix, and also removing full blocks going out of the suffix. In detail, a pruned client is downloading all the headers, then, by using them, it checks proofs-of-work and linking structure(or parent id only?). Then it downloads a UTXO snapshot for some height from its peers. Finally, full blocks after the snapshot are to be downloaded and applied to get a current UTXO set. A pruned fullnode is also skipping AD-transformation block part, like a fullnode. Additional setting: "suffix" - how much full blocks to store(w. some minimum set?). Its regular modifiers processing is the same as for fullnode regime, while its bootstrap process is different:

1. Send ErgoSyncInfo message to connected peers.
2. Get response with INV message, containing ids of blocks, better than our best block.
3. Request headers for all ids from 2.
4. On receiving header:

```
if(History.apply(header).isSuccess) {
    if(!(localScore == networkScore)) GOTO 1
    else GOTO 5
 } else {
    blacklist peer
 }
```

5. Request historical UTXOManifest for at least BlocksToKeep back.
6. On receiving UTXOSnapshotManifest:

```
UTXOSnapshotManifest.chunks.foreach { chunk =>
    request chunk from sender() //Or from random fullnode
  }
```

7. On receiving UTXOSnapshotChunk:

```
State.applyChunk(UTXOSnapshotChunk) match {
     case Success(Some(newMinimalState)) => GOTO 8
     case Success(None) => stay at 7
     /*we need more chunks to construct state.
     TODO periodicaly request missed chunks*/
     case Failure(e) => ???
     //UTXOSnapshotChunk or constcucted state roothash is invalid
  }
```

8. Request BlockTransactions starting from State we have

```
History.headersStartingFromId(State.headerId).foreach { header =>
    send message(GetBlockTransactionsForHeader(header)) to Random fullnode
  }
```

9. On receiving BlockTransactions: same as in Fullnode.7 .
10. Operate as Fullnode.

Light Full-Node Mode

This mode is based on an idea to use a 2-party authenticated dynamic dictionary built on top of UTXO set. A light-fullnode holds only a root digest of a dictionary. It checks all the full blocks, or some suffix of the full blockchain, depending on setting, thus starting from a trusted pre-genesis digest or some digest in the blockchain. A light-fullnode is using AD-transformations (authenticated dictionary transformations) block section containing batch-proof for UTXO transformations to get a new digest from an old one. It also checks all the transactions, but doesn't store anything but a single digest for that. Details can be found in the paper https://eprint.iacr.org/2016/994.

Additional settings : "depth" - from which block in the past to check transactions (if 0, then go from genesis).

"additional-checks" - light-fullnode trusts previous digest and checks current digest validity by using the previous one as well as AD-transformations.

"additional-depth" - depth to start additional checks from.

1. Send ErgoSyncInfo message to connected peers.
2. Get response with INV message, containing ids of blocks, better than our best block.
3. Request headers for all ids from 2.
4. On receiving header:

```
if(History.apply(header).isSuccess) {
    if(!(localScore == networkScore)) GOTO 1
    else GOTO 5
 } else {
    blacklist peer
 }
```

5. Request BlockTransactions and ADProofs starting from BlocksToKeep back in History (just 1 last header after node botstrapping):

```
History.lastBestHeaders(BlocksToKeep).foreach { header =>
    send message(GetBlockTransactionsForHeader(header)) to Random fullnode
    send message(GetAdProofsHeader(header)) to Random fullnode
  }
```

6. On receiving modifier BlockTransactions or ADProofs:

```
if(History.apply(modifier) == Success(ProgressInfo)) {
  /* TODO if history now contains both ADProofs and BlockTransactions,
  it should return ProgressInfo with both of them, otherwise
  it should return empty ProgressInfo */
if(State().apply(ProgressInfo) == Success((newState, ADProofs)))
{
if("mode"=="pruned-full") drop BlockTransactions and ADProofs older than BlocksToKeep
}
else {
          /*Drop Header from history, because it's transaction sequence is not valid*/
          History.drop(BlockTransactions.headerId)
     }
  }
```

Light-SPV Mode

This mode is not about checking any full blocks. Like in Bitcoin, an SPV node is downloading block headers only, and so checks only proofs of work and links. Unlike Bitcoin's SPV, the Light-SPV is downloading and

checking not all the headers but a sublinear(in blockchain length) number of them(in benchmarks, this is about just tens of kilobytes instead of tens or hundreds of megabytes for Bitcoin/Ethereum). Light-SPV mode is intended to be useful for mobile phones and low-end hardware.

Bootstrap

1. Send GetPoPoWProof for all connections.
2. On receive PoPoWProof apply it to History (History should be able to determine, whether this PoPoWProof is better than it's current best header chain).
3. GOTO regular regime.

Regular

1. Send ErgoSyncInfo message to connected peers
2. Get response with INV message, containing ids of blocks, better than our best block.
3. Request headers for all ids from 2.
4. On receiving header:

```
if(History.apply(header).isSuccess) {
   State.apply(header) // just change state roothash
   if(!isInitialBootstrapping) Broadcast INV for this header
} else {
   blacklist peer
}
```

Mode-Related Settings

Ergo has the following settings determines a mode:

- ADState: Boolean - keeps state roothash only.
- VerifyTransactions: Boolean - download block transactions and verify them (requires BlocksToKeep == 0 if disabled).
- PoPoWBootstrap: Boolean - download PoPoW proof only
- BlocksToKeep: Int - number of last blocks to keep with transactions, for all other blocks it keep header only. Keep all blocks from genesis if negative
- MinimalSuffix: Int - minimal suffix size for PoPoW proof (may be pre-defined constant).

'if(VerifyTransactions == false) require(BlocksToKeep == 0)' Mode from "multimode.md" can be determined as follows:

Ergo Block Structure

Ergo block consists of 4 parts:

- *Header* - minimal amount of data required to synchronize the chain and check PoW correctness. Also contains hashes of other sections.
- *BlockTransactions* - sequence of transactions, included in this block.
- *ADProofs* - proofs for transactions included into the corresponding BlockTransactions section of a block. Allows light clients to verify all the transactions and calculate new root hash.
- *Extension* - additional data, that does not correspond to previous sections. Contains interlinks and current parameters of the chain (when an extension belongs to a block at the end of the voting epoch).

Header

Field	Size	Description
version	1	block version, to be increased on every soft- and hardfork
parentId	32	id of parent block
ADProofsRoot	32	hash of ADProofs for transactions in a block
stateRoot	32	root hash (for an AVL+ tree) of a state after block application
transactionsRoot	32	root hash (for a Merkle tree) of transactions in a block
timestamp	8	block timestamp(in milliseconds since beginning of Unix Epoch)
nBits	8	current difficulty in a compressed form
height	4	block height
extensionRoot	32	root hash of extension section
powSolution	75-107 (depends on d)	solution of Autolykos PoW puzzle
votes	3	votes for changes in system parameteres, one byte per vote

Some of these fields may be calculated by node by itself if it is in a certain mode:

- parentId: if(status==bootstrap AND PoPoWBootstrap == false).
- ADProofsRoot: if(status==regular AND ADState==false AND BlocksToKeep>0).
- stateRoot: if(status==regular AND ADState==false AND BlocksToKeep>0).

Extension

Extension is a key-value storage for a variety of data.

A key is always 2-bytes long, maximum size of a value is 64 bytes. Extension could be no more than of $16,384$ bytes. Some keys have predefined semantics. In particular, if the first byte of a key equals to $0x00$, then the second byte defines parameter identifier, and the value defines value of the parameter. See Section 9.1 for details. Another predefined key is used for storing interlinks vector - the first byte of the key

is $0x01$, the second one corresponds to index of the link in the vector and the value contains actual link (32 bytes) prefixed with the number of times it appears in the vector (1 byte). Other prefixes may be used freely.

Ergo Modifiers Processing

This section describes processing algorithm for Ergo modifiers in all security modes. Unlike most of blockchain systems, Ergo have the following types of modifiers: In-memory:

1. In-memory:
   - Transaction - in-memory modifier.
   - TransactionIdsForHeader - ids of transactions in concrete block.
   - UTXOSnapshotManifest - ids of UTXO chunks and
2. Persistent:
   - BlockTransactions - Sequence of transactions, corresponding to 1 block.
   - ADProofs - proof of transaction correctness relative to corresponding UTXO.
   - Header, that contains data required to verify PoW, link to previous block, state root hash and root hash to it's payload (BlockTransactions, ADProofs, Interlinks ...).
   - UTXOSnapshotChunk - part of UTXO.
   - PoPoWProof

   Ergo will have the following parameters, that will determine concrete security regime:

   - ADState: Boolean - keep state roothash only.
   - VerifyTransactions: Boolean - download block transactions and verify them (requires BlocksToKeep == 0 if disabled).
   - PoPoWBootstrap: Boolean - download PoPoW proof only.
   - BlocksToKeep: Int - number of last blocks to keep with transactions, for all other blocks it keep header only. Keep all blocks from genesis if negative.
   - MinimalSuffix: Int - minimal suffix size for PoPoW proof (may be pre-defined constant).

     ```
     if(VerifyTransactions == false) require(BlocksToKeep == 0)
     ```

   Mode from "multimode.md" can be determined as follows:

   ```
   mode = if(ADState == false && VerifyTransactions == true
   && PoPoWBootstrap == false && BlocksToKeep < 0) "full"
   else if(ADState == false && VerifyTransactions == true
   && PoPoWBootstrap == false && BlocksToKeep >= 0) "pruned-full"
   else if(ADState == true && VerifyTransactions == true
   && PoPoWBootstrap == false) "light-full"
   else if(ADState == true && VerifyTransactions == false
   && PoPoWBootstrap == true && BlocksToKeep == 0) "light-spv"
   else if(ADState == true && VerifyTransactions == true
   && PoPoWBootstrap == true && BlocksToKeep == 0) "light-full-PoPoW"
   else //Other combinations are possible
   ```

Modifiers processing

```
def updateHeadersChainToBestInNetwork() = {
  1.2.1. Send ErgoSyncInfo message to connected peers
  1.2.2. Get response with INV message,
  containing ids of blocks, better than our best block
  1.2.3. Request headers for all ids from 1.2.2.
  1.2.4. On receiving header
```

```
   if(History.apply(header).isSuccess) {
      if(!(localScore == networkScore)) GOTO 1.2.1
   } else {
      blacklist peer
      GOTO 1.2.1
   }
}
```

bootstrap

Download headers:

```
if(PoPoW) {
 1.1.1. Send GetPoPoWProof(suffix = Max(MinimalSuffix ,BlocksToKeep)) for all connections
 1.1.2. On receive PoPoWProof apply it to History
  /*
  History should be able to determine,
  whether this PoPoWProof is better, than it's current best header chain */
} else {
  updateHeadersChainToBestInNetwork()
}
```

Download initial State to start process transactions:

```
if(ADState == true) {
  Initialize state with state roothash from block header BlocksToKeep ago
} else if(BlocksToKeep < 0 || BlocksToKeep > History.headersHeight) {
  Initialize state with genesis State
} else {
/*
We need to download full state BlocksToKeep back in history
TODO what if we can download state only "BlocksToKeep - N"
or "BlocksToKeep + N" blocks back?
*/
  2.1. Request historical UTXOSnapshotManifest for at least BlocksToKeep back
  2.2. On receiving UTXOSnapshotManifest:
    UTXOSnapshotManifest.chunks.foreach ( chunk => request chunk from sender()
/*Or from random fullnode*/
  2.3. On receiving UTXOSnapshotChunk
  State.applyChunk(UTXOSnapshotChunk) match {
     case Success(Some(newMinimalState)) => GOTO 3
     case Success(None) => stay at 2.3
     /*we need more chunks to construct state. TODO periodicaly request missed chunks*/
     case Failure(e) => ???
     /*UTXOSnapshotChunk or constcucted state roothash is invalid*/
  }
}
```

Update State to best headers height:

```
 if(State.bestHeader == History.bestHeader) {
    //Do nothing, State is already updated
  } else if(VerifyTransactions == false) {
/*Just update State rootshash to best header in history*/
    State.setBestHeader(History.bestHeader)
```

```
  } else {
/*we have headers chain better than full block */
    3.1.
      assert(history contains header chain from State.bestHeader to History.bestHeaders)
      History.continuation(from = State.bestHeader, size = ???).get.foreach { header =>
        sendToRandomFullNode(GetBlockTransactionsForHeader(header))
        if(ADState == true) sendToRandomFullNode(GetADProofsForHeader(header))
      }
    3.2. On receiving modifiers ADProofs or BlockTransactions
      /*TODO History should return non-empty ProgressInfo
      only if it contains both ADProofs and BlockTransactions,
      or it contains BlockTransactions and ADState==false*/
      if(History.apply(modifier) == Success(ProgressInfo)) {
        if(State().apply(ProgressInfo) == Success((newState, ADProofs))) {
          if(ADState==false) ADProofs.foreach ( ADProof => History.apply(ADProof))
          if(BlocksToKeep>=0)
          /*remove BlockTransactions and ADProofs older than BlocksToKeep from history*/
        } else {
      /*Drop Header from history,
      because it's transaction sequence is not valid*/
          History.drop(modifier.headerId)
        }
      } else {
        blacklistPeer
      }
      GOTO 3
    }
```

GOTO regular mode.

Regular

Two infinite loops in different threads with the following functions inside:

1. UpdateHeadersChainToBestInNetwork()
2. Download and update full blocks when needed

```
 if(State.bestHeader == History.bestHeader) {
    //Do nothing, State is already updated
  } else if(VerifyTransactions == false) {
    //Just update State rootshash to best header in history
    State.setBestHeader(History.bestHeader)
  } else {
    //we have headers chain better then full block
    3.1. Request transaction ids from all headers without transactions
      assert(history contains header chain from State.bestHeader to History.bestHeaders)
      History.continuation(from = State.bestHeader, size = ???).get.foreach { header =>
        sendToRandomFullNode(GetTransactionIdsForHeader(header))
        if(ADState == true) sendToRandomFullNode(GetADProofsForHeader(header))
      }
    3.2. On receiving TransactionIdsForHeader:
      Mempool.apply(TransactionIdsForHeader)
```

```
    TransactionIdsForHeader.filter(txId => !MemPool.contains(txId)).foreach { txId =>
      request transaction with txId
    }
  3.3. On receiving a transaction:
    if(Mempool.apply(transaction).isSuccess) {
       Broadcast INV for this transaction
       Mempool.getHeadersWithAllTransactions { BlockTransactions =>
          GOTO 3.4 //now we have BlockTransactions
       }
    }
  3.4. (same as 3.2. from bootstrap)
}
```

Economic properties

Emission Rules

Ergo emission will last for 2080799 blocks (8 years with planned 2 minute block interval) — for the first 525600 blocks (2 years) 75 Erg will be issued per block and after that the block reward will be reduced by 3 Erg every 64800 blocks (3 months). To fund the development, during the first 655200 blocks (2.5 years) the part of the block rewards that exceeds 67.5 will go to foundation treasury instead of a miner. (see Fig. 1).

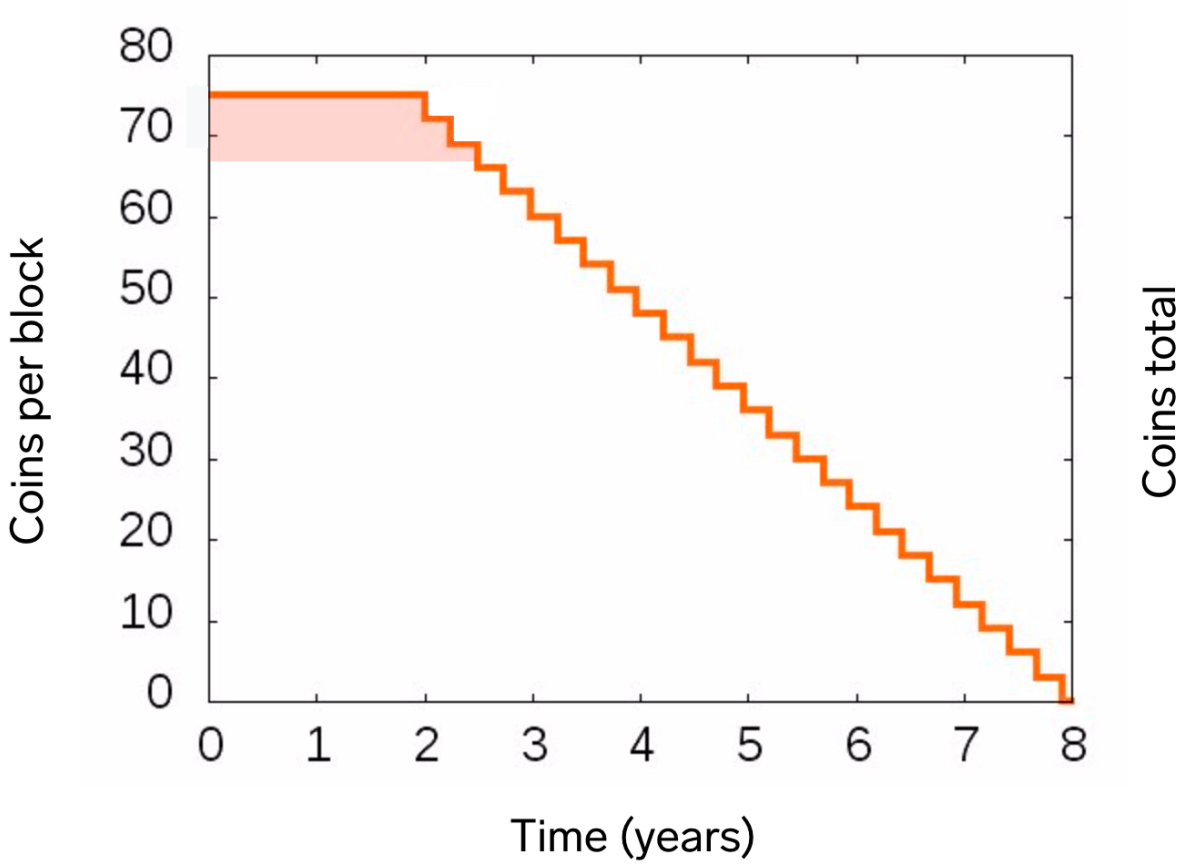

Figure 1: Ergo emission rule

Instead of having an implicit emission rule via a special type of transaction (e.g. coinbase transaction in Bitcoin), Ergo coins emission rule is defined explicitly by sigma-state transactional language.

Total miners rewards of 93409132.5 Erg will be created in the genesis state in a box, protected by a script defined at https://git.io/fhgOq. This script allows a miner to take only a part of all coins every block. Transaction that spends this output should have exactly 2 outputs: the first one should have the same protecting script and contain all input coins minus current block reward, while the second one may be collected by the current block miner after at least 720 blocks delay.

Total foundation rewards will be kept in the genesis state in a box with 4330791.5 Erg and will be protected protected by the script defined at https://git.io/fhoqw. First output of the transaction that spends this box should at least have the value of the remaining treasury. In addition, conditions from R4 register of this box should be satisfied, allowing to protect this output from undesirable spent. At the beginning R4 register will contain 2-of-3 multisignature proposition with the hardcoded public keys.

Ergo emission will start from zero with no premine. As proof-of-no-pre-mine genesis state will contain an

additional box with 1 Erg inside protected by unspendable proposition. This box registers R4-R6 will contain latest news headlines from The Guardian, Vedomosti, and Xinhua, while registers R7 and R8 will contain latest block id's from Bitcoin and Ethereum blockchains correspondingly.

Storage rent

We outline following two properties required for long-term survivability of the chain:

- coins protected by keys being lost should be returned into circulation. Otherwise, after the end of the emission period, amount of the coins in the circulation will always decrease and eventually reach zero.
- nothing should be kept in the state forever and for free. Otherwise, the size of the state is always increasing with time, thus reducing clients performance.

To achieve this, we propose the following storage fee consensus rules. Register $R3$ of a box contains tuple $(creation\_height, tx\_id||out\_num)$, where $creation\_height$ provided by a user is used to determine the block height at the moment of transaction generation. Transaction can only be put in the block of height h if for every created box its $creation\_height \leq h$.

Once the subsidized period for the box ends (that is, $current\_block\_height \geq box.creation\_height + SP$, see SP value below), anyone (presumably, a miner) can create a new box out with the exactly the same content (including the guarding script) except the monetary value and $R3$ contents. The monetary value is to reduced by $K \cdot B$ maximum, where B is the spent box ($self$) size and K is the storage cost for the period SP. Thus, the difference is to be paid to the miner. If box value is less that the storage fee, all the box content including tokens could be spent by the miner.

For efficient lookup (to find a proper output regarding an expired input without an iteration over transaction outputs), we require a spending proof for an expired box to be just a context extension which contains only an index of an output which is trying to spend the box. The variable identifier for the index in the extension is 127.

We propose the following concrete parameters:

- SP - length of a period for which box can be stored in a State for free untouched. It is a constant, namely, $SP = 1051200 \approx 4$ years.
- K - cost of storage of 1 byte of data in a State for the period of SP blocks. Should be determined by miner votes, 1250000($nanoErg/SP$) by default, max value is 2500000.

Transactions

ErgoTransaction is an atomic state transition operation. It destroys box from the state and creates new ones. If transaction is spending boxes protected by some non-trivial scripts, its inputs should also contain proof of spending correctness - context extension (user-defined key-value map) and data inputs (links to existing boxes in the state) that may be used during script reduction to crypto, signatures that satisfies the remaining cryptographic protection of the script. Transactions are not encrypted, so it is possible to browse and view every transaction ever collected into a block.

Box Format

A box is made of registers (and nothing but registers), we allow every box in the system to have up to 10 registers. We denote the registers as $R_0, R_1, \ldots, R_9$. From these registers, four are filled with mandatory values: R_0 contains monetary value of a box, R_1 contains serialized guard script, R_2 contains tokens, R_3 contains declared creation height, unique identifier of transaction which created the coin and also an index of the box in the transaction.

Each register is an expression in sigma language. Thus the registers are typed: every register contains a value of some type. Types are defined in . The value should be evaluated (i.e. it should be a concrete constant value, not a function of a known output type).

We introduce $extract()$ function, which is reading contents of a register, for example, $extract(c, R_0)$ extracts monetary value from the box c.

Box bytes used to obtain the box identifier, build authenticated tree for the state, and build a transaction, are to be formed as follows:

- *monetary value.* We use VLQ-encoded unsigned long value to store monetary value of the box
- *bytes of a script.* To see how the script is serialized, see (). The script is to be represented as register R1 by wrapping its bytes are constant array of constant bytes.
- *creation height* VLQ-encoded unsigned integer value
- *number of tokens* which box is carrying. Represented as a one-byte unsigned integer.
- *tokens.* A box can carry multiple tokens. A single record in this field is represented as a tuple $token\_identifier -> amount$, where identifier is of 32 bytes and amount is VLQ-encoded integer.
- *number of additional registers.* Extra registers should come in order (R4, ..., etc), so this number, represented as 1-byte unsigned value, defines how much non-empty registers are coming after mandatory ones. If the number is zero, next field is missed.
- *additional registers.* Extra registers are serialized without any delimiters. Each register is serialized as a sigmaexpression.
- *transaction identifier.* 32-bytes long identifier of a transaction which created the box
- *index of a transaction output.* VLQ-encoded index of the box in the transaction outputs.

Box candidate

Here we describe difference between a box and a box candidate. A box has a unique identifier to be defined deterministically from its contents. Thus we need to have different identifiers for boxs of the same meaning, even if they are created by the same transaction. We also require a box to have an identifier which is derived solely from box contents, thus we can not use *(transaction_id, output_id)* pair as Bitcoin Core implementation is doing.

To solve the issue we split concepts of a box and a box candidate. A box candidate is defining semantics of the corresponding box i.e. has the same values for all the registers except of the register R_3 which is set to be $(creation\_height, transaction\_id||output\_index)$ for a box and $(creation\_height, 0^{34*8})$ (so instead of real transaction id and output index, a zero-bit string of 34 bytes is used) for a box candidate.

Transaction Format

Ergo transaction consists of 3 parts:

- *inputs* - links to boxs that will be removed from the state during the transaction application. Every inputs consists of box id, proof for final sigma proposition of this box protecting script and a context extension to be used during script evaluation.
- *data inputs* - links to boxs that won't be removed from the state, but will be available in context of regular inputs scripts.
- *outputs* - new boxs that will be included into the state during the transaction application.

Transaction bytes are to be formed as follows:

- *inputs number* - VLQ-encoded number of inputs.
- *inputs* - every input is represented as 32-byte id of a box to be spent, VLQ-encoded length of signature, signature itself, VLQ-encoded number of key-value pairs of context extension, and context extension pairs, that are 1-byte key and values serialized as a sigmaexpression
- *number of data inputs* - VLQ-encoded number of data inputs.
- *data inputs* - 32-byte length ids of data boxes
- *distinct tokens number* - number of distinct tokens in the transaction, represented as 1 byte unsigned integer.
- *token ids* - 32-byte length ids of tokens in the transaction
- *outputs number* - VLQ-encoded number of transaction outputs
- *boxcandidates* - every coin candidates are serialized in the same way, as box bytes from 8.1 section, but transaction identifier and index of a transaction output are missed, and also tokens are represented as pairs $index -> amount$, where index is 1-byte index of token in token ids section

Inputs signatures should sign a $bytesToSign(tx)$ message, that is calculated as transaction bytes, as if all it's signatures are empty (and thus VLQ-encoded length of signature equals to 0). Transaction id is calculated as a Blake2b256 hash from message to sign.

Transaction Merkle Tree

Like a miner in the Bitcoin protocol is building a Merkle tree of block transactions, as well as a Merkle tree of transaction witnesses (after the Segwit upgrade), in Ergo, a miner should build a Merkle tree (and include a correct root hash of the tree into a block header), which is in case of Ergo combines both transactions and their respective spending proofs.

This tree is to be constructed as follows. A data block under a leaf of the tree could be empty or 64 bytes in length. Data of 64 bytes contains identifier of the transaction (256-bits digest of transaction bytes without spending proofs) and 256 bits of a digest of all the spending proofs for the transaction combined. Data for $i-th$ transaction in the block (starting from 0) is authenticated by the $i-th$ leaf. A leaf is $hash(0||pos||data)$, if the *data* is not empty (we do add prefix for domain separation), or *null* otherwise. Here, *pos* is a position of the transaction in the block. For internal nodes, a node is $hash(1||left\_child||right\_child)$, if either left child or right child of the node is not *null*, *null* othewise. If root hash is *null*, we are writing all zeros (of hash function output length) instead of it.

Signing A Transaction

To spend a box a spending transaction tx has as an input, one needs to use $bytesToSign(tx)$ (note that different inputs can be signed in parallel, however, the coins being spent are to be specified before signing), as well as current state of the blockchain, or *context*. The signer also can extend the context by putting values there.

By having this data, a signer (or a prover) of an input first reduces the guarding proposition for the input box by evaluating, it using the context. Possible results of the reduction are:

- abort if estimated upper-bound cost of evaluation (and verification) is too high.
- true means that the box is spendable by anyone
- false means that the box is not spendable at all (at least according to the current context)
- expression still containing predicates over the context. That means context is not enough to evaluate some predicates over it. Prover can look into its own not revealed yet secrets in order to extend context. If the secrets are found, prover is reducing the expression further and doing the next step, if there is nothing more to evaluate. Otherwise the prover aborts.
- expression containing only expressions over secret information provable via Σ-protocols. This is the most common case, and we are going to explain it in details further.

We are having possible complex expression, like $dlog(x_1) \vee (dlog(x2)/\ dlog(x3))$, where $dlog(x)$ means "prove me knowledge of a secret w, such as for a known group with generator g, $g^w = x$, via a non-interactive form of the Schnorr protocol". Internally, this exression is represented as a tree (). This proof is to be proven and then serialized into a binary spending proof (which could be included into a transaction) as follows:

Proving steps for a tree:

1. bottom-up: mark every node real or simulated, according to the following rule. DLogNode – if you know the secret, then real, else simulated. $\vee$: if at least one child real, then real; else simulated. $\wedge$: if at least one child simulated, then simulated; else real. Note that all descendants of a simulated node will be later simulated, even if they were marked as real. This is what the next step will do.

Root should end up real according to this rule – else you won't be able to carry out the proof in the end.

2. top-down: mark every child of a simulated node "simulated." If two or more more children of a real $\vee$ are real, mark all but one simulated.

3. top-down: compute a challenge for every simulated child of every $\vee$ and $\wedge$, according to the following rules. If $\vee$, then every simulated child gets a fresh random challenge. If $\wedge$ (which means $\wedge$ itself is simulated, and all its children are), then every child gets the same challenge as the $\wedge$.

4. bottom-up: For every simulated leaf, simulate a response and a commitment (i.e., second and first prover message) according to the Schnorr simulator. For every real leaf, compute the commitment (i.e., first prover message) according to the Schnorr protocol. For every $\vee/\wedge$ node, let the commitment be the union (as a set) of commitments below it.

5. Compute the Schnorr challenge as the hash of the commitment of the root (plus other inputs – probably the tree being proven and the message).

6. top-down: compute the challenge for every real child of every real $\vee$ and $\wedge$, as follows. If $\vee$, then the challenge for the one real child of $\vee$ is equal to the XOR of the challenge of $\vee$ and the challenges for all the simulated children of $\vee$. If $\wedge$, then the challenge for every real child of $\wedge$ is equal to the the challenge of the $\wedge$. Note that simulated $\wedge$ and $\vee$ have only simulated descendants, so no need to recurse down from them.

Now, how to get a flat binary string containing (e, z) pairs (challenge and prover's response for a leafsub-protocol) from the tree:

Transaction Validation Rules

We split validation of a single transaction into two stages. There are *statelesschecks* which could be done by the transaction only being presented. Stateful checks are requiring knowledge of the current state, in the form of the whole UTXO set or a part of it, namely concrete boxes a transaction is destroying along with proof of authenticity for them (against a root hash included into a last block header).

Stateless checks are:

- a transaction must spend at a least one input and create at least one output. A transaction spends no more than 32767 inputs and creates no more than 32767 outputs.
- all the output amounts must be non-negative.
- an output can not contain more than 4 assets. All the assets amounts must be positive.
- transaction outputs collectively could not contain more than 16 assets.

Stateful checks are:

- all the input boxes are members of the UTXO set or where created by other transactions of this block.
- all the data input boxes are members of the UTXO set or where created by other transactions of this block.
- total input and output ergo amounts must be the same. Adding up should be done with overflow checks.
- all the inputs have valid spending proofs.
- total transaction cost which consists of cost of all spending proofs verification plus the cost of all tokens containing in transaction inputs and outputs validation (calculated as $(\sum_{i=1}^{n_{in}} m_i + m_{tx} + \sum_{i=1}^{n_{out}} k_i + k_{tx}) * e$, where n_{in} - number of inputs, m_i - number of tokens in i'th input, m_{tx} - number of tokens in all inputs in total, n_{out} - number of outputs, k_i - number of tokens in i'th output, k_{tx} - number of tokens in all outputs in total, e - cost of accessing a single token (adjustable via miners voting)) should not be greater than a limit per block (which is adjustable via miners voting as well).
- for each kind of asset in the outputs, total output amount for the asset should be no more than the total input amount, if the asset identifier is not equals to identifier of the first input; otherwise, the total output amount must be positive. The latter case corresponds to issuance of a new asset, while the former sets asset preservation rule. Please note that for total input amount of an asset we do not require for strict equality of the input and output amounts: the output amount could be less than input amount (or zero).

Unified Transactions

Full-Block Validation Rules

Below are rules for block validation when a node is verifying transactions ($VerifyTransactions = 1$)

- every transaction in a block is referencing to inputs from the UTXO set or created by previous transactions in the block. Please note that it is not possible for an input to refer to an output of some follow-up transaction of the block.
- every transaction in a block is valid (see Section 8.6 for transaction validation rules).
- total cost of validation of all the inputs of all the transactions in the block is no more than allowed limit.
- root hash of the authenticated UTXO set after application of the block transactions is the same as in the header.
- for a node keeping UTXO, hash of the calcualted state transformations proof is the same as announced in the header of the block.
- header is valid .

Token Emission

Token emission is incorporated with just one conservation law added, in addition to the standard one (that is, sum of monetary values for transaction inputs should equal the corresponding sum for outputs). One output can hold tokens of multiple kinds, the maximum number of tokens per output is 4 (in addition to the main Ergo token). They are stored in the register R3 as a sequence of $\{token\_id : amount\}$ tuples. This is the only kind of data which can be stored in R3. The emission is organized as appending an item to the dictionary. To avoid collisions, appended $token\_id$ must equal id of the first input of the generating transaction. The uniqueness of outputs yields the uniqueness of tokens. Obviously, only one output can contain a new asset, and a transaction may create no more than one new asset.

The validation script is then

$$\forall\, id \in \{i \,|\, \exists\, out \in outputs : i \in out.R3.keys\}$$
$$\left(\sum_{in \in inputs} in.R3[id] = \sum_{out \in outputs} out.R3[id]\right) \vee (id = inputs[0].id)\ .$$

Here $\sum$ stands for the safe sum, which ensures non-negativeness of all the values, and the absence of integer overflows on the way. The controlled emission of the tokens may be organized by attaching the emission script to the output which contains newly generated $token\_id$.

Voting

Many parameters can be changed on-the-fly via miners voting, such as instructions costs, computational cost limit per block, block size limit, storage fee factor, block version, and so on. Voting for the block version (so for a soft-fork) lasts for 32 epochs (see epoch length below), and requires more than 90 percent of the miners to vote for the change. For less critical changes (such as block size limit), simple majority is enough. We will further refer to the changes of the first kind as to foundational changes, and we refer to the changes of the second kind as to everyday changes. Per block, a miner can vote for two everyday changes and also one foundational change, with the votes to be included into the header of the block.

To vote "Yes" ("I'm agree on the change proposed"), and also to propose change in the first block of an epoch, a miner is publishing identifier of the change directly in the block header. To vote "No" (or avoid voting at all, which is the same), a miner is simply writing zero value instead of a corresponding byte (another option is to provide a vote identifier which is not being considered within the epoch).

System constants:

- Voting epoch length = 1024 blocks.
- Voting epochs per foundational change = 32
- Voting epochs before approved foundational change activation = 128

Parameters table

The following table describes vote identifiers, default values (during launch), possible steps, and also minimum and maximum values. If the step is not defined in the table, its value is defined as $\max(\lfloor current\_value/100 \rfloor, 1)$. If minimum value for a parameter is not defined, it equals to zero. If maximum value is not defined, it equals to 1,073,741,823.

To propose or vote for increasing a parameter, a miner is inluding a parameter identifier (id) into the block header. If the miner is for decreasing parameter, he is including ($-id$) into the block header.

Id	Description	Default	Step	Min	Max
1	Storage fee factor (in nanoErgs per byte per storage period)	1250000	25000	0	5000000
2	Minimum monetary value of a box (in nanoErgs)	360	10	0	10000
3	Maximum block size	524288	-	16384	-
4	Maximum cumulative computational cost of a block	1000000	-	16384	-
5	Validation cost of accessing a single token	100	-	-	-
6	Validation cost per one transaction input	2000	-	-	-
7	Validation cost per one data input	100	-	-	-
8	Validation cost per one transaction output	100	-	-	-

Parameter values are to be written into extension section on first block of a voting epoch, that is, in the extension of a block when its $height \, mod \, 1024 = 0$. Parameters for initial moment of time ($height = 1$) are simply hardcoded.

Proposing a change and voting for it

To propose a change, in the first block of a voting epoch (of $1,024$ blocks, so in a block of $height \, mod \, 1024 = 0$), a miner is posting an identifier of a vote for a change. There are three slots (three bytes) in a block header for changes to propose, with two slots reserved for everyday changes and the third one for proposing a softfork. Slot not occuppied by a proposal is to be set to zero. Votes could come in any order. Examples of the bytes: $(0, 1, 120)$, $(120, -3, 0)$. In the first case, a miner is proposing to increase storage fee factor ($id : 1$), and also proposes a soft-fork ($id : 120$), In the second case, a miner is proposing to decrease block size ($id : -3$), and also is proposing a soft-fork ($id : 120$).

To vote for a proposal (which is, again, proposed in the first block of an epoch) within the epoch, a miner is including vote identifier into the block header. Identifiers not proposed in the first block of the epoch are ignored.

If majority of votes within an epoch are supporting an everyday change (so at least 513 blocks are containing an identifier), a new value of the parameter should be written into the extension section of the first block of the next epoch.

Voting for a soft-fork

A soft-fork happens when a protocol version supported by the network is being increased. This version is written in a block header. Semantics behind versioning is not defined ahead of time by the protocol and so up to clients and their developers. The protocol version is being written into every block header, with initial value during launch to be set to 1.

The protocol version upgrade is to be done in the following steps:

1. A miner is proposing to increase protocol version.
2. Other miners are voting within 32 epochs for the proposal
3. If the soft-fork proposal is being rejected (so if it gathers no more than 90% of votes during voting period, i.e. no more than $\lfloor 32 * 1024 * 90/100 \rfloor = 29491$ votes), new voting may be proposed next epoch after the voting done.
4. If the soft-fork proposal is being approved, activation period of 128 epochs is starting. The first block after the activation period is called activation height. It is prohibited to vote for a new soft-fork during the activation and also on the activation height. Soft-fork data is still to be written to corresponding extension sections (see below) during the activation period, and on activation height. Block version written into extensions is to be increased from the first block of the activation period, while protocol version in headers is still the same. Protocol version in headers is increased from the activation height.

To start voting for a soft-fork, a miner needs to publish the identifier 120 in the first block of the epoch, consider, for example, that its height is h_s. Next epoch, a miner should post height when start fork was

proposed $(122 : h_s)$ in the extension section of the block, and number of votes collected in the previous epochs v_s should be written there as well as $(121 : v_s)$.

Examples:

- Assume that a vote for soft-fork is proposed in block number 1024 (by writing a record into the extension with a key equals to 120 and any value). Assume that within this epoch, so before block number 2048, 500 votes were collected. Thus a miner which is generating block #2048 must write height when the soft-fork voting has been started in the extension section of the block as (122 : 1024 key-value pair), as well as number of votes supporting the fork gathered within the epoch done (in form of 121 : 500). Assume that the new epoch started with the block #2048 brings 600 votes for fork. Then a miner generating the block #3072 must write down the following pairs there: $(121, 1100)$ and $(122, 1024)$.
- The voting is to be done on height $1024 + 1024 * 32 = 33792$, so the last vote could be written into the block #33791. Block #33792 still should contain correct values for votes collected and voting starting height.
- Lets assume that voting for the soft-fork was not successful, i.e. no more than 29491 votes for soft-fork have been gathered between blocks #1024 (inclusive) and #33792 (exclusive). Then the next epoch block, so the block #34816 should clear the old voting parameter values. A new voting for soft-fork may be started in this new epoch, so in the block #34816, if the block contains a vote for the soft-fork (i.e. a record with key = 120 in the extension section), then the block must contain new starting height and votes collected values (i.e. following pairs: (121 : 0) and (122 : 34816)). If there is no new voting started, all the parameters regarding soft-fork voting must be cleared off of the block #34816.
- Now assume that the voting for the soft-fork was successful, i.e. more than 29491 votes for soft-fork have been gathered between blocks #1024 (inclusive) and #33792 (exclusive). In this case, activation epoch is starting immediately after voting has been done, so on height 33792. Activation epoch lasts for 128 epochs, so the first block after activation period has height of 164864. Please note that any vote for a soft-fork during the activation period is prohibited. Also, on a first block of a voting epoch soft-fork voting parameters should be written (height when voting had been started and also number of votes collected). On the first block after activation, protocol version written into a block must be increased. The first block after activation should carry soft-fork voting parameters. The parameters must be cleared next epoch, so in a block #165888. A new voting may be started in the block #165888, similarly to the case of non-successful voting.

Proof-of-Work

Part III
The Reference Implementation

Blockchain synchronization

Ergo modifiers can be in one of the following states:

- *Unknown* - synchronization process for corresponding modifier is not started yet.
- *Requested* - modifier was requested from another peer.
- *Received* - modifier was received from another peer, but is not applied to history yet.
- *Held* - modifier was held by NodeViewHolder - PersistentModifiers are held by History, Ephemeral modifiers are held by Mempool.
- *Invalid* - modifier is permanently invalid.

The goal of the synchronization process is to move modifiers from *Unknown* to *Held* state. In the success path modifier changes his statuses *Unknown->Requested->Received->Held*, however if something went wrong (e.g. modifier was not delivered) it goes back to *Unknown* state (if node is going to receive it in future) or to *Invalid*state (if node is not going to receive this modifier anymore).

From *Unknown* to *Requested*

Modifier can go from *Unknown* state to *Requested* one by different ways, that depend on current node status (bootstrapping/stable) and modifier type.

Inv protocol

Inv (inventory) message contains a pair: *(ModifierTypeId, Seq[ModifierId])*. When one node sends *Inv* message to another one, it is assumed that this node contains modifiers with the specified ids and type and is ready to send on request.

Node broadcasts *Inv* message in 2 cases:

- - When it successfully applies a modifier to *History* and modifier is new enough. This is useful to propogate new modifiers as fast as possible when nodes are already synced with the network
- - When it receives *ErgoSyncInfo* message (see **Headers synchronization** for more details)

When node received *Inv* message it

- - filter out modifiers, that are not in state *Unknown*
- - request remaining modifiers from the peer that sent *Inv* message. Modifier goes into *Requested* state.

Headers synchronization

First, node should synchronize it's headers chain with the network. In order to achieve this every *syncInterval* seconds node calculates *ErgoSyncInfo* message, containing ids of last *ErgoSyncInfo.MaxBlockIds* headers and send it to peers, defined by function *peersToSyncWith()*. If there are outdated peers (peers, which status was last updated earlier than *syncStatusRefresh* seconds ago) *peersToSyncWith()* return outdated peers, otherwise it returns one random peer which blockchain is better and all peers with status Unknown .

To achieve more efficient synchronization, node also sends *ErgoSyncInfo* message every time when headers chain is not synced yet, but number of requested headers is small enough (less than $desiredInvObjects/2$).

On receiving *ErgoSyncInfo* message, node calculates *OtherNodeSyncingStatus*, which contains node status (*Younger*, *Older*, *Equal*, *Nonsense* or *Unknown*) and extension - *Inv* for next *maxInvObjects* headers missed by *ErgoSyncInfo* sender. After that node sends this *Inv* message to *ErgoSyncInfo* sender.

Block section synchronization

After headers application, a node should synchronize block sections (BlockTransactions, Extension and ADProofs), which amount and composition may vary on node settings (node with UTXO state does not need to download ADProofs, node with non-negative *blocksToKeep* should download only block sections for last *blocksToKeep* blocks, etc.).

In order to achieve this, every *syncInterval* seconds node calculate *nextModifiersToDownload()* - block sections for headers starting at height of best full block, that are in *Unknown* state. These modifiers are requested from random peers (since we does not know a peer who have it), [1]. and they switch to state *Requested*.

To achieve more efficient synchronization, node also requests *nextModifiersToDownload()* every time when headers chain is already synced and number of requested block sections is small enough (less than $desiredInvObjects/2$).

When headers chain is already synced and node applies block header, it return *ProgressInfo* with *ToDownload* section, that contains modifiers our node should download and apply to update full block chain. When NVS receives this ToDownload request, it requests these modifiers from random peers and these modifiers goes to state *Requested*.

From *Requested* to *Received*

When our node requests a modifier from other peer, it puts this modifier and corresponding peer to special map *requested* in *DeliveryTracker* and sends *CheckDelivery* to self with *deliveryTimeout* delay.

When a node receives modifier in *requested* map (and peer delivered this modifier is the same as written in *requested*) - NVS parse it and perform initial validation. If modifier is invalid (and we know, that this modifier will always be invalid) NVS penalize peer and move modifier to state *Invalid*. If the peer have provided incorrect modifier bytes (we can not check, that these bytes corresponds to current id) penalize peer and move modifier to state *Unknown*. If everything is fine, NVS sends modifier to NodeViewHolder(NVH) and set modifier to state *Received*.

When *CheckDelivery* message comes, node check for modifier - if it is already in *Received* state, do nothing. If modifier is not delivered yet, node continue to wait it up to *maxDeliveryChecks* times, and after that penalize peer (if not requested from random peer) and stop expecting after that (modifier goes to *Unknown* state).

From *Received* to *Held*

When NVH receives new modifiers it puts these modifiers to modifiersCache and then applies as much modifiers from cache as possible. NVH publish *SyntacticallySuccessfulModifier* message for every applied modifier and when NVS receives this message it moves modifier to state *Held*. If after all applications cache size exceeds size limit, NVH remove outdated modifiers from cache and publish *ModifiersProcessingResult* message with all just applied and removed modifiers. When NVS receives *ModifiersProcessingResult* message it moves all modifiers removed from cache without application to state *Unknown*.

[1] we can keep a separate modifierId->peers map for modifiers, that are not received yet and try to download from this peers first

Wallet

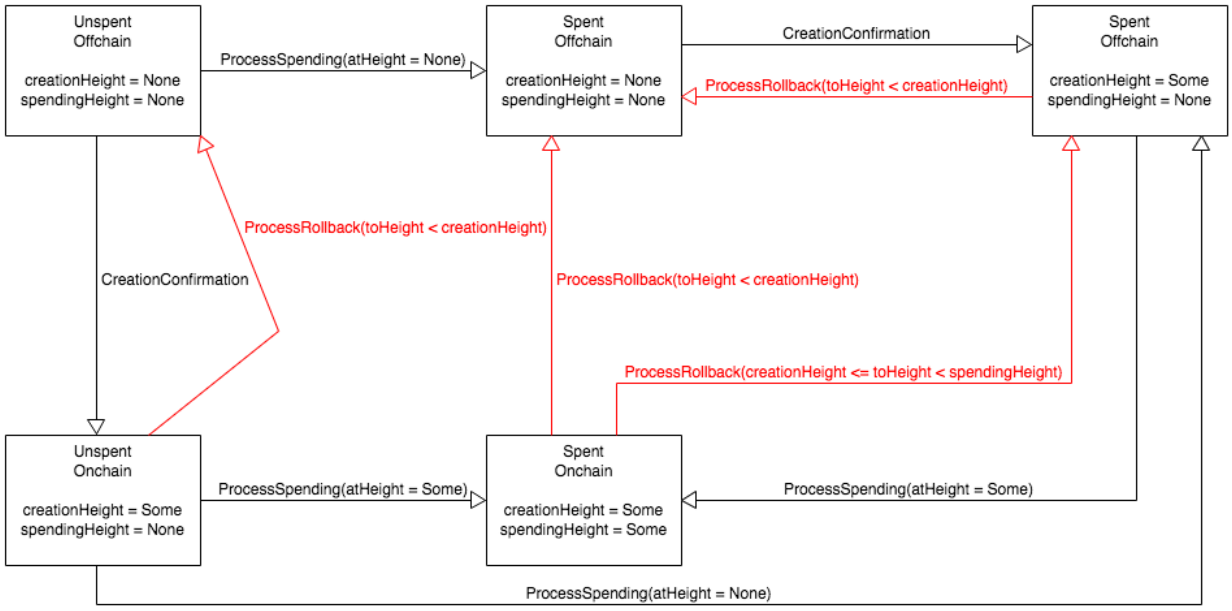

Figure 2: State transitions for boxes tracked by the wallet