Multi-Stage Contracts in the UTXO Model

Alexander Chepurnoy, Amitabh Saxena

Ergo Platform
{kushti,amitabh123}@protonmail.ch

Abstract. Smart contract platforms such as Bitcoin and Ethereum allow writing programs that run on a decentralized computer. Bitcoin uses short-lived immutable data structures called UTXOs for data manipulation. Ethereum, on the other hand uses, long-lived mutable data structures called *accounts*. UTXOs are easier to handle, less error prone and scale better because the only operation we can do with them is to create or destroy (i.e., spend) them. The code inside a UTXO is executed only once, when it is spent. Additionally, this code refers to only local context (i.e., it is stateless). In Ethereum's account based system, there is a shared global context which each account can access and modify, thereby causing side affects. However, the benefit of persistent storage offered by accounts makes up for these drawbacks. In this work, we describe how to emulate persistent storage in UTXO based systems using a technique called *transaction trees*. This allows us to emulate the functionality of account-based systems such as Ethereum without the overhead of accounts. We demonstrate this via several examples which include contracts for a Rock-Paper-Scissors game, crowdfunding and an initial coin offering (ICO). The contracts are created in a UTXO based smart contract platform called Ergo that supports transaction trees.

1 Introduction

Smart contracts were envisioned in 1994 by Nick Szabo [1], a legal scholar and cryptographer. He proposed the concept of self-executing contracts written in executable code and stored in a replicated manner on distributed computers that enforced the rules written in the code. Bitcoin [2] can be seen as the first implementation of this concept using a fully decentralized ledger whose contracts primarily pertain to transfer and store of value, i.e., as a currency system. Ethereum [3] is an example of a general-purpose smart contract platform.

The limited application of Bitcoin allows optimizations focussed on long-term survivability and scalability. Firstly, all data and code is stored in short-lived immutable objects (called UTXOs [4]). A user can execute code inside a UTXO by supplying some input (which may contain additional code). A UTXO is destroyed once its code is executed (i.e., it is spent). Secondly, all computation is performed within a *local context*; any code pertaining to a UTXO can only operate on data for that UTXO and does not have access to the global state.

In contrast, Ethereum follows a different set of design principles in which the code and data is contained in long-lived mutable objects called *accounts*. This was done because UTXOs are stateless and do not provide persistent storage. Not only can Ethereum code modify data in its own account, but also trigger execution of code in other accounts. Thus, Ethereum code operates over a *shared global context* representing all existing accounts.

The results of [5] allow UTXO-based systems to emulate Ethereum-like functionality by reducing the computation to Rule-110 [6,7]. However, such reductions are not very efficient and a more practical solution for the same is desirable. In particular, we need higher-level abstractions (instead of Rule-110) that enable UTXO-based systems to efficiently emulate Ethereum functionality and maintain Bitcoin's scalability. In this work we describe a technique called *transaction trees* that allow writing advanced smart contracts in UTXO based systems. As proof of concept, we implemented such contracts on a UTXO-based platform called Ergo that supports transaction trees.

Context Enrichment. In Bitcoin and other existing UTXO systems, the context is just the UTXO being processed. In order for a UTXO-based system to support transaction trees, the context must be rich enough to contain at least the entire spending transaction. More formally, for any UTXO based blockchain, we can define the following levels of context, each extending the previous:

1. The current UTXO plus the blockchain height and time
2. The current spending transaction (other inputs and outputs)
3. The current block's solution.
4. The current block (other sibling transactions)

Any platform at Level 2 and above is suitable for transaction trees. In this regard, Bitcoin operates at Level 1 and Ergo at Level 3. Note that in Level 4 we cannot check validity of transactions independently of other transactions in the block. Hence it is more complex to implement Level 4.

In this work we show via examples how to create efficient Ethereum-like contracts in the UTXO model using transaction trees. The examples include a Rock-Paper-Scissors game, an Initial Coin Offering (ICO) campaign and a new primitive called *reversible addresses* for securely storing funds.

2 Ergo Overview

The Ergo platform is a Level 3 UTXO based blockchain that allows general-purpose smart contracts via a highly expressible language called ErgoScript. Since Ergo follows the UTXO based model, all data and code is stored in immutable objects called *boxes*. As in Bitcoin, a transaction in Ergo can spend (destroy) multiple boxes and create new ones. A box is made of up to ten *registers* labelled $R_0, R_1, \ldots R_9$, four of which are mandatory. R_0 contains the monetary value, R_1 contains the *guard script*, R_2 contains assets (tokens) and R_3 contains a unique identifier of 34 bytes made up of a transaction ID and an output index. The guard script in R_1 encodes a spending condition, which must be satisfied for spending the box. Deploying a contract involves creating an unspent box with the relevant ErgoScript code in R_1 and populating other registers if necessary. The contract is executed by spending the box.

Similar to Bitcoin, an ErgoScript program also cannot access the global state and all computation must be done only using a local context. Unlike Bitcoin, this context is quite rich and allows access to the entire spending transaction [8]. In particular, an ErgoScript program defines the spending condition using predicates on the inputs and outputs of the transaction interleaved with Sigma protocols [9]. Thus, an ErgoScript program can enforce the spending transaction's structure (such as requiring that assets are transferable only to a certain address). Additionally, the program can require the spender to prove knowledge of the discrete logarithm of some public value using a protocol called `proveDlog`, which is based on the Schnorr identification scheme [9]. All public keys (such as `alice` and `bob`) in the following sections are of type `proveDlog`. Similar to Bitcoin, a Pay-to-Script-Hash (P2SH) address in Ergo contains the hash of a script that must be provided when spending from that address. The script encodes the actual spending condition. Ergo also supports Pay-to-Script (P2S) address, where the actual script is encoded in the address.

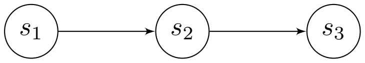
One useful feature of Ethereum is the ability to store and access a large amount of data, which Ergo also provides. However, Ergo contracts do not store the actual data in the blockchain. Rather, the data is stored off chain and a short digest is stored in the blockchain. A user wishing to access or modify this data must provide correct proofs of (non)-existence or modification for this digest, as in the ICO example of Section 4.3.

3 Transaction Trees

A powerful feature of ErgoScript is the ability to specify the spending transaction's structure in a fine-grained manner. Among the many things we can specify, the important ones are: (1) the number of input and output boxes, (2) the value of any box, and (3) the guarding script of any box. This allows us to create *transaction trees*, where the contract in an input box requires an output box to contain some predefined contract, thereby ensuring that only a certain sequence of contracts are possible. We will use this to convert an Ethereum-style long-lived contract into multi-stage contracts in the UTXO model, where each stage encodes data and code to be carried over to the next stage.

Transaction Chains: Before describing transaction trees, we describe a simpler primitive called transaction chains. A transaction chain is used for creating a multi-stage protocol whose code does not contain loops or 'if' statements. A transaction chain is created as follows:

1. Represent an Ethereum contract's execution using n sequential steps, where each step represents a transaction that modifies its state. The states before and after a transaction are the start and end nodes respectively of a directed graph, with the transaction as the edge joining them. As an example, a 3-stage contract, such as the ICO example of Section 4.3 is represented as:



The states contain data and the code that was executed in the transaction.

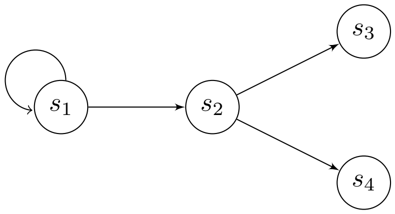
2. Hardwire state n's code and data inside state $n-1$'s code. Then require the code of state $n-1$ to output a box containing state n's code and data. An example is given in the following pseudocode:

   ```
   out.propositionBytes == state_n_code &&
   out.R4[Int].get == SELF.R4[Int].get // ensure data is propagated
   ```

   The above code uses the field `propositionBytes` of a box, which contains the binary representation of its guard script as collection of bytes.
3. Repeat Step 2 by replacing $(n, n-1)$ by $(n-1, n-2)$ while $n > 2$.

To avoid code size increase at each iteration, we should ideally work with hashes, as in `hash(out.propositionBytes) == state_n_code_hash`. However, for clarity of presentation, we will skip this optimization.

Transaction Trees: A transaction tree is an extension of transaction chains where the code can contain 'if' statements and *simple loops*, i.e., where some start and end nodes are the same. The following figure illustrates a transaction tree.



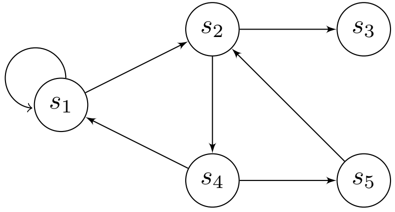
An 'if' statement is handled using the following pseudocode.

```
if (condition) { out.propositionBytes == state_3_code }
else { out.propositionBytes == state_4_code }
```

A simple loop is a special case of the 'if' statement:

```
if (condition) { out.propositionBytes == state_2_code }
else { out.propositionBytes == SELF.propositionBytes }
```

Transaction graphs: Ergo supports a more advanced technique called *transaction graphs*, where cycles are allowed in contract references, as shown below.



Discussion of such contracts is beyond the scope of this work and we refer the reader to [10, Section 3.3.3] for an example of such a contract. All the examples in this paper are based on transaction trees.

4 Multi-Stage Contracts

4.1 Reversible Addresses

An example of multi-stage contract is a *reversible address*, which has anti-theft features in the following sense: any funds sent to a reversible address can only be spent in way that allows payments to be reversed for a certain time. To motivate this feature, consider managing the hot-wallet of an exchange or mining pool used for handling customer withdraws. A hot-wallet is an address for which the private key is stored on the server. Such addresses are necessary for facilitating automated withdrawals. Being a hot-wallet, its private key is susceptible to compromise and funds being stolen. We want to ensure that we are able to recover any stolen funds in the event of such a compromise, provided that the breach is discovered within, say, 24 hours of the first unauthorized withdraw.

Assume that `alice` is the public key of the hot-wallet and `carol` is the public key of the trusted party. The private key of `carol` will be needed for reversing payments and must be stored offline. Let `b` be the estimated number of blocks in a 24 hour period. Let Bob with public key `bob` be a customer wishing to withdraw funds, which will be paid out by the hot-wallet.

In Ethereum, we can do this by sending funds to an account having with a contract C_b that allows `carol` to withdraw funds at least `b` blocks and after that they can only be withdrawn by `bob`. We could use the same account (contract instance) for multiple withdraws by Bob, but the optimal way is to have a new account for each withdraw, emulating the UTXO model. The funds for this must also come from another account with a contract C_a that ensure that withdraw can only be done to a contract with the structure of C_b.

In Ergo, this is done by a two-stage protocol, where the second stage implements C_b and the first stage implements C_a. The following script called `withdrawScript` implements the second stage. This will be the guarding script of the hot-wallet's withdraw transaction paying to `bob`.

```
val bob = SELF.R4[SigmaProp].get    // public key of customer withdrawing
val bobDeadline = SELF.R5[Int].get  // max locking height
(bob && HEIGHT > bobDeadline) || (carol && HEIGHT <= bobDeadline)
```

This above script is referenced in the first stage script given next.

```
val isChange = {(b:Box) => b.propositionBytes == SELF.propositionBytes}
val isWithdraw = {(b:Box) =>
    b.R5[Int].get >= HEIGHT + blocksIn24h &&
    b.propositionBytes == withdrawScript
}
alice && OUTPUTS.forall({(b:Box) => isChange(b) || isWithdraw(b)})
```

The reversible address is the P2SH address of the above script. Any funds sent to this address are subject to the withdraw rules that we desire. In the normal case, Bob will spend the box after roughly `blocksIn24h` blocks. If an unauthorized transaction from the hot-wallet is detected, an abort procedure is triggered using the private key of `carol` and funds in any unspent boxes sent from the hot-wallet are diverted to a secure address. Note that the trusted party (`carol`) is bound to the hot-wallet address. A new address is needed for a different trusted party.

Although such addresses are designed for securing hot-wallet funds, the may have other applications. One example is for automated-release escrow payments in online shopping, where `carol` can be the public key of any mutually agreed adjudicating party.

4.2 Rock-Paper-Scissors Game

Our next example of a multi-stage contract is the Rock-Paper-Scissors game, which is often used to introduce Ethereum [11]. The game is played between two players, Alice and Bob. Each player chooses a secret independently and the game is decided after the secrets are revealed. Let $a, b \in \mathbb{Z}_3$ be the secrets of Alice, Bob respectively, with the understanding that (0, 1, 2) represent (rock, paper, scissors). If $a = b$ then the game is a draw, otherwise if $a - b \in \{1, -2\}$ then Alice wins else Bob wins.

The first party to reveal the secret has a disadvantage, since the other party can adaptively choose and win. In the real world, both parties reveal their secrets simultaneously to prevent this. In the virtual world, however, this cannot be enforced. Hence this attack must be handled using *cryptographic commitments*, where the first party, Alice, does not initially reveal her secret, but rather only a commitment to that secret. The modified game using commitments is as follows:

1. Alice commits to her secret a by inputting her commitment $c = Comm(a)$.
2. Bob inputs his public value b. At this stage, Alice knows if she won or lost.
3. Alice opens her commitment and reveals a, after which the winner is decided.

This works fine assuming that Alice is *well-behaved*, i.e., she always opens her commitment irrespective of whether she won or lost. In the real world, however, we also need to consider the possibility that Alice never opens her commitment. Border cases such as these make smart contracts quite tricky, because once deployed, it is not possible to add "bug-fixes" to them. In this example, we must penalize Alice (with a loss) if she does not open her commitment within some stipulated time.

The complete game is coded in ErgoScript in two stages. In the first stage, Alice creates a *start-game* box that encodes her game rules. In the second stage, Bob spends the start-game box and creates two *end-game* boxes spendable by the winner. These new boxes indicate that the game has ended.

To start the game, Alice decides a game amount x (of Ergo's primary token), which each player must contribute. She then selects a secret s and computes a commitment $c = H(a||s)$ to a. Finally, she locks up x tokens along with her commitment c inside the start-game box protected by the following script:

```
OUTPUTS.forall(
  {(out:Box) =>
    val b = out.R4[Byte].get
    val bobDeadline = out.R6[Int].get
    bobDeadline >= HEIGHT+30 && out.value >= SELF.value &&
    (b == 0 || b == 1 || b == 2) &&
    out.propositionBytes == outScript
  }
) && OUTPUTS.size == 2 && OUTPUTS(0).R7[SigmaProp].get == alice &&
OUTPUTS(0).R4[Byte].get == OUTPUTS(1).R4[Byte].get // same b
```

The above code requires that the spending transaction must create exactly two outputs, one paying to each player in the event of a draw or both paying to the winner otherwise. In particular, the code requires that (1) register R_7 of the first output must contain Alice's public key (for use in the draw scenario), (2) register R_4 of each output must contain Bob's choice, and (3) each output must contain at least x tokens protected by `outScript`, which is given below:

```
val s = getVar[Coll[Byte]](0).get  // Alice's secret byte string s
val a = getVar[Byte](1).get  // Alice's secret choice a
val b = SELF.R4[Byte].get    // Bob's public choice b
val bob = SELF.R5[SigmaProp].get // Bob's public key
val bobDeadline = SELF.R6[Int].get // after this, Bob wins by default
val drawPubKey = SELF.R7[SigmaProp].get
val valid_a = (a == 0 || a == 1 || a == 2)
val validCommitment = blake2b256(s ++ Coll(a)) == c
val validAliceChoice = valid_a && validAliceChoice
val aliceWins = (a - b) == 1 || (a - b) == -2
val receiver = if (a == b) drawPubKey else (if (aliceWins) alice else bob)
(bob && HEIGHT > bobDeadline) || (receiver && validAliceChoice)
```

The above code protects the two end-game boxes that Bob generates. The condition `(bob && HEIGHT > bobDeadline)` guarantees that if Alice does not open her commitment before a certain deadline, then Bob automatically wins. Note that Bob has to ensure that R_7 of the second output contains his public key. Additionally, he must ensure that R_5 of both outputs contains his public key (see below). We don't encode these conditions because if Bob doesn't follow the protocol, he will automatically lose.

4.3 Initial Coin Offering

Another popular use-case of Ethereum is an Initial Coin Offering (ICO) contract. An ICO mirrors an Initial Public Offering (IPO) and provides a mechanism for a project to collect funding in some tokens and then issue "shares" (in the form of some other tokens) to investors. Generally, an ICO comprises of 3 stages:

1. *Funding:* During this period, investors are allowed to fund the project.
2. *Issuance:* A new asset token is created and issued to investors.
3. *Withdrawal:* Investors can withdraw their newly issued tokens.

Compared to the previous examples, our ICO contract is quite complex, since it involves multiple stages and parties. The number of investors may run into thousands, and the naive solution would store this data in the contract, as in the ERC-20 standard [12]. Unlike Ethereum, Ergo does not permit storing large datasets in a contract. Rather, we store only a 40-bytes header of (a key, value) dictionary, that is authenticated like a Merkle tree [13]. To access some elements in the dictionary, or to modify it, a spending transaction should provide lookup or modification proofs. This allows a contract to authenticate large datasets using very little storage and memory.

Funding: The project initiates the ICO by creating a box with the guard script given below. The box also contains a authenticating value for an empty dictionary of (investor, balance) pairs in R_5, where investor is the hash of a script that will guard the box with the withdrawn tokens (once the funding period ends).

```
val selfIndexIsZero = INPUTS(0).id == SELF.id
val proof = getVar[Coll[Byte]](1).get
val toAdd = INPUTS.slice(1, INPUTS.size).map({(b: Box) =>
    val pk = b.R4[Coll[Byte]].get
    val value = longToByteArray(b.value)
    (pk, value)
})
val modifiedTree = SELF.R5[AvlTree].get.insert(toAdd, proof).get
val expectedTree = OUTPUTS(0).R5[AvlTree].get
val selfOutputCorrect =
  if (HEIGHT < 2000) OUTPUTS(0).propositionBytes == SELF.propositionBytes
  else OUTPUTS(0).propositionBytes == issuanceScript
val outputsCorrect = OUTPUTS.size == 1 && selfOutputCorrect

selfIndexIsZero && outputsCorrect && modifiedTree == expectedTree
```

The first funding transaction spends this box and creates a box with the same script and updated data. Further funding transactions spend the box created from the previous funding transaction. The box checks that it is first input of each funding transaction, which must have other inputs belonging to investors. The investor inputs contain a hash of the withdraw script in register R_4. The script also checks (via proofs) that hashes and monetary values of the investing inputs are correctly added to the dictionary of the new box, which must be only output with the correct amount of ergs (we ignore fee in this example). In this stage, which lasts at least till height 2,000, withdraws are not permitted and ergs can only be put into the project. The first transaction with height of 2,000 or more should keep the same data but change the output's script called `issuanceScript` described next.

Issuance: This stage requires only one transaction to get to the next stage (the withdrawal stage). The spending transactions makes the following modifications. Firstly, it changes the list of allowed operations on the dictionary from "inserts only" to "removals only". Secondly, the contract checks that the proper amount of ICO tokens are issued. In Ergo, each transaction can issue at most one new kind of token, with the (unique) identifier of the first input box. The issuance contract checks that a new token is issued with amount equal to the nano-ergs collected till now. Thirdly, the contract checks that a spending transaction is indeed re-creating the box with the guard script corresponding to the next stage, the withdrawal stage. Finally, the contract checks that the spending transaction has 2 outputs (one for the project tokens and one for the ergs withdrawn by the project). The complete script is given below.

```
val openTree = SELF.R5[AvlTree].get
val closedTree = OUTPUTS(0).R5[AvlTree].get
val correctDigest = openTree.digest == closedTree.digest
val correctKeyLength = openTree.keyLength == closedTree.keyLength
val removeOnlyTree = closedTree.enabledOperations == 4
val correctValue = openTree.valueLengthOpt == closedTree.valueLengthOpt
val tokenId: Coll[Byte] = INPUTS(0).id
val tokenIssued = OUTPUTS(0).tokens(0)._2
val correctTokenNumber = OUTPUTS(0).tokens.size == 1 &&
                         OUTPUTS(1).tokens.size == 0
val correctTokenIssued = SELF.value == tokenIssued
val correctTokenId = OUTPUTS(0).R4[Coll[Byte]].get == tokenId &&
                     OUTPUTS(0).tokens(0)._1 == tokenId
val valuePreserved = OUTPUTS.size == 2 && correctTokenNumber &&
                     correctTokenIssued && correctTokenId
val stateChanged = OUTPUTS(0).propositionBytes == withdrawScript
val treeIsCorrect = correctDigest && correctValue &&
                    correctKeyLength && removeOnlyTree

projectPubKey && treeIsCorrect && valuePreserved && stateChanged
```

Withdrawal: Investors are now allowed to withdraw ICO tokens under a guard script whose hash is stored in the dictionary. Withdraws are done in batches of N. A withdrawing transaction, thus, has $N+1$ outputs; the first output carries over the withdrawal sub-contract and balance tokens, and the remaining N outputs have guard scripts and token values as per the dictionary. The contract requires two proofs for the dictionary elements: one proving that values to be withdrawn are indeed in the dictionary, and the second proving that the resulting dictionary does not have the withdrawn values. The complete script called `withdrawScript` is given below

```
val removeProof = getVar[Coll[Byte]](2).get
val lookupProof = getVar[Coll[Byte]](3).get
val withdrawIndexes = getVar[Coll[Int]](4).get
val tokenId: Coll[Byte] = SELF.R4[Coll[Byte]].get
val withdrawals = withdrawIndexes.map({(idx: Int) =>
    val b = OUTPUTS(idx)
    if (b.tokens(0)._1 == tokenId)
       (blake2b256(b.propositionBytes), b.tokens(0)._2)
    else
       (blake2b256(b.propositionBytes), 0L)
})
val withdrawValues = withdrawals.map({(t: (Coll[Byte], Long)) => t._2})
val total = withdrawValues.fold(0L, {(l1: Long, l2: Long) => l1 + l2 })
val toRemove = withdrawals.map({(t: (Coll[Byte], Long)) => t._1})
val initialTree = SELF.R5[AvlTree].get
val removedValues = initialTree.getMany(toRemove, lookupProof).map(
    {(o: Option[Coll[Byte]]) => byteArrayToLong(o.get)}
)
val valuesCorrect = removedValues == withdrawValues
val modifiedTree = initialTree.remove(toRemove, removeProof).get
val outTreeCorrect = OUTPUTS(0).R5[AvlTree].get == modifiedTree
val selfTokenCorrect = SELF.tokens(0)._1 == tokenId
val outTokenCorrect = OUTPUTS(0).tokens(0)._1 == tokenId
val outTokenCorrectAmt = OUTPUTS(0).tokens(0)._2 + total == SELF.tokens(0)._2
val tokenPreserved = selfTokenCorrect && outTokenCorrect && outTokenCorrectAmt
val selfOutputCorrect = OUTPUTS(0).propositionBytes == SELF.propositionBytes

outTreeCorrect && valuesCorrect && selfOutputCorrect && tokensPreserved
```

Note that the above ICO example contains many simplifications. For instance, we don't consider fee when spending the project box. Additionally, the project does not self-destruct after the withdraw stage. We refer the reader to [14] for the full example.

5 Conclusion

We gave examples demonstrating that, despite being UTXO-based, Ergo can support complex multi-stage contracts found in Ethereum. In particular, we described:

1. A Rock-Papers-Scissors game with provable fairness (Section 4.2).
2. Reversible Addresses having anti-theft features (Section 4.1).
3. A full featured ICO that accepts funding in ergs (Section 4.3).

The examples used the idea of *transaction trees* to emulate persistent storage by linking several UTXOs containing small pieces of code to form a large multi-stage protocol. We refer the reader to ErgoScript repository and tutorials [8,10] for additional examples of multi-stage contracts, including a Local Exchange Trading Systems (LETS), non-interactive mixing, atomic swaps and many more.

References

1. Nick Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.
2. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.
3. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
4. Jameson Lopp. Unspent transactions outputs in Bitcoin. `http://statoshi.info/dashboard/db/unspent-transaction-output-set`, accessed Nov 7, 2016.
5. Alexander Chepurnoy, Vasily Kharin, and Dmitry Meshkov. Self-reproducing coins as universal turing machine. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 57–64, Cham, 2018. Springer International Publishing.
6. Matthew Cook. A concrete view of rule 110 computation. *Electronic Proceedings in Theoretical Computer Science*, 1:31–55, Jun 2009.
7. Turlough Neary and Damien Woods. P-completeness of cellular automaton rule 110. In *International Colloquium on Automata, Languages, and Programming*, pages 132–143. Springer, 2006.
8. Ergoscript, a cryptocurrency scripting language supporting noninteractive zero-knowledge proofs. `https://docs.ergoplatform.com/ErgoScript.pdf`, 03 2019.
9. Ivan Damgård. On Σ-Protocols, 2010. `http://www.cs.au.dk/~ivan/Sigma.pdf`.
10. Advanced ergoscript tutorial. `https://docs.ergoplatform.com/sigmastate_protocols.pdf`, 03 2019.
11. Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. *IACR Cryptology ePrint Archive*, 2015:460, 2015.
12. The Ethereum Wiki. Erc20 token standard. `https://theethereum.wiki/w/index.php/ERC20_Token_Standard`, 2018.
13. Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 376–392, Cham, 2017. Springer International Publishing.
14. Alexander Chepurnoy. An ico example on top of ergo. `https://ergoplatform.org/en/blog/2019_04_10-ico-example/`, 2019.