Advanced ErgoScript Tutorial

Ergo Developers

April 23, 2019

Abstract

Ergo is a smart contract platform based on Bitcoin's UTXO model and Ethereum-like functionality that it provides via a language called ErgoScript. The syntax of ErgoScript is a subset of Scala's. In this article, we give a high-level overview of ErgoScript using examples.

We use ErgoScript to create smart contracts for several protocols such as an XOR game, a rock-paper-scissors game, *reversible addresses* that have anti-theft features, and *ErgoMix*, a privacy enhancing protocol, which can be considered a non-interactive variant of CoinJoin.

1 Introduction

A key feature of ErgoScript is the use of *Sigma-Protocols* (written Σ-protocols)[1] interleaved with predicates on the transaction and the blockchain state. ErgoScript currently supports two such-Protocols defined on a group G of prime order q, written here in multiplicative form. The first, denoted as `proveDlog(u)`, is a *proof of knowledge of Discrete Logarithm* of some arbitrary group element u with respect to a fixed generator g, where the spender proves knowledge of x such that $u = g^x$. This is derived from Schnorr signatures [2]. The second, denoted as `proveDHTuple`, is a *proof of knowledge of Diffie-Hellman Tuple* and is explained in Section 3.3.

The main structure in ErgoScript is a *box*, which is roughly like a UTXO of Bitcoin. A transaction spends (destroys) some boxes by using them as inputs and creates new boxes as outputs. ErgoScript is used to write the *spending condition* protecting funds stored in a box. The spender of a box must provide a 'proof' of satisfying that condition.

The following sections present smart contracts using ErgoScript. More details of ErgoScript are available in the white paper [3] and the code for the below examples is available on GitHub [4].

2 Basic Examples: Enhanced Spending Contracts

The examples below use P2SH address and highlight some limitation of Bitcoin.

2.1 Short-lived Unconfirmed Transactions: Paying for Coffee

Alice is paying for coffee using cryptocurrency. She makes a payment but it is taking a long time for the transaction to confirm. She decides to pay using cash and leave. However, she is worried that her original payment will eventually confirm and then she will either lose it or have to ask for a refund. In Bitcoin, she can try to double spend the transaction, which is not always guaranteed, even if using *replace-by-fee*. ErgoScript has a better solution using *timed-payments* so that if the

transaction is not confirmed before a certain height, it is no longer valid. Timed-payments require Alice's funds be stored in a *timed address*, which is the P2SH of the following script:

```
alice && HEIGHT <= getVar[Int](1).get
```

Here `alice` is a *named constant* representing her public key. Any funds deposited to this address can only be spent if the spending transaction satisfies following:

1. Context variable with id 1 of the box being spent must contain an integer, say i.
2. The height at mining should be less than or equal to i.

Observe that if the transaction is not mined before height i then the transaction becomes invalid. When paying at a coffee shop, for example, Alice can set i close to the height h at the time of broadcast, for instance, $i = h + 10$. Alice can still send non-timed payments by making i very large. Since the context variables are part of the message in constructing the zero-knowledge proof, a miner cannot change it (to make this transaction valid).

2.2 Hot-Wallet Contracts: Reversible Addresses

We create a useful primitives called *reversible addresses*, designed for storing funds in a hot-wallet. Any funds sent to a reversible address can only be spent in way that allows payments to be reversed for a certain time. The idea was proposed for Bitcoin [5] (using the moniker *R-addresses*) and requires a hardfork. In ErgoScript, however, this can be done natively.

To motivate this feature, consider managing the hot-wallet of a mining pool or an exchange. Funds withdrawn by customers originate from this hot-wallet. Being a hot-wallet, its private key is succeptible to compromise. One day you discover several unauthorized transactions from the hot-wallet, indicating a breach. You wish there was a way to reverse the transactions and cancel the withdraws but alas this is not the case. In general there is no way to recover the lost funds once the transaction is mined, even if the breach was discovered within minutes.

We would like that in the event of such a compromise, we are able to save all funds stored in this wallet and move them to another address, provided that the breach is discovered within a specified time (such as 24 hours) of the first unauthorized withdraw.

To achieve this, we require that all coins sent from the hot-wallet (both legitimate and by the attacker) have a 24 hour cooling-off period, during which the created boxes can only be spent by a trusted private key that is was selected *before* the compromise occurred. This trusted key must be different from the hot-wallet private key and should ideally be in cold storage. After 24 hours, these boxes become 'normal' and can only be spent by the receiver.

This is done by storing the hot-wallet funds in a special type of address denoted as *reversible*. Assume that `alice` is the public key of the hot-wallet and `carol` is the public key of the trusted party. Note that the trusted party must be decided at the time of address generation and cannot be changed later. To use a different trusted party, a new address has to be generated. Let `blocksIn24h` be the estimated number of blocks in a 24 hour period. A reversible address is a P2SH address whose script encodes the following conditions:

1. This input box can only be spent by `alice`.
2. Any output box created by spending this input box must have in its register R_5 a number at least `blocksIn24h` more than the current height.

3. Any output box created by spending this input box must be protected by a script requring the following:

   (a) Its register R_4 must have an arbitrary public key called `bob`.

   (b) Its register R_5 must have an arbitrary integer called `bobDeadline`.

   (c) It can only be spent spent by `carol` if `HEIGHT` $\leq$ `bobDeadline`.

   (d) It can only be spent by `bob` if `HEIGHT` $>$ `bobDeadline`.

Thus, all funds sent from such addresses have a temporary lock of `blocksIn24h` blocks. This can be replaced by any other desired value but it must be decided at the time of address generation. Let `bob` be the public key of a customer who is withdrawing. The sender (`alice`) must ensure that register R_4 of the created box contains `bob`. In the normal scenario, `bob` will be able to spend the box after roughly `blocksIn24h` blocks (with the exact number depending on `bobDeadline`).

If an unauthorized transaction from `alice` is detected, an "abort procedure" is triggered via `carol`: all funds sent from `alice` and in the locked state are suspect and need to diverted elsewhere.

To create a reversible address, first create a script, `withdrawScript`, with the following code:

```
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)
```

Let `feeProposition` be the script of a box that pays mining fee and `maxFee` be the maximum fee allowed in one transaction. The reversible address is the P2SH address of the following script:

```
val isChange = {(out:Box) => out.propositionBytes == SELF.propositionBytes}
val isWithdraw = {(out:Box) =>
  out.R5[Int].get >= HEIGHT + blocksIn24h &&
  out.propositionBytes == withdrawScript
}
val isFee = {(out:Box) => out.propositionBytes == feeProposition}
val isValid = {(out:Box) => isChange(out) || isWithdraw(out) || isFee(out)}

val totalFee = OUTPUTS.fold(0L, {
    (x:Long, b:Box) => if (isFee(b)) x + b.value else x
  }
)
alice && OUTPUTS.forall(isValid) && totalFee <= maxFee
```

2.3 Cold-Wallet Contracts: Limiting Spending Capacity

Assume an address is protected by 2 private keys, corresponding to the public keys `alice` and `bob`. For security, we want the following conditions to hold:

1. One key can spend at most 1% or 100 Ergs (whichever is higher) in one day.

2. If both keys are spending then there are no restrictions.

Let `blocksIn24h` be the number of blocks in 24 hours. Instead of hardwiring 1% and 100 Ergs, we will use the named constants `percent` and `minSpend` respectively. The cold-wallet address is the P2SH address of the following script:

```
val storedStartHeight = SELF.R4[Int].get // block at which the period started
val creationHeight = SELF.creationInfo._1 // creation height
val startHeight = min(creationHeight, storedStartHeight)
val notExpired = HEIGHT - startHeight <= blocksIn24h // expired if 24 hrs passed
val min = SELF.R5[Long].get // min Balance needed in this period

val ours:Long = SELF.value - SELF.value * percent / 100
val keep = if (ours > minSpend) ours else 0L // topup should keep min >= keep
val nStart:Int = if (notExpired) start else HEIGHT
val nMin:Long = if (notExpired) min else keep

val out = OUTPUTS(0)
val valid = INPUTS.size == 1 && out.propositionBytes == SELF.propositionBytes &&
  out.value >= nMin && out.R4[Int].get >= nStart && out.R5[Long].get == nMin})

(alice && bob) || ((alice || bob) && min >= keep && (nMin == 0 || valid))
```

Spending from this address is done in periods of 24 hours or more such that the maximum spendable is a fixed fraction of the amount at the beginning of the period. We do this by requiring the spending transaction to have an output with value greater than the minumum (which is stored in R_5) and paying back to the same address. The start of the current period is stored in R_4. Both registers are copied to the new output within the same period and get new values for if the current period has expired.

3 Two-party Protocols

We focus on two-round, two-party protocols. In the first round, the first party, Alice, initiates the protocol by creating a box protected by a script encoding the protocol rules. In the second round, the second party, Bob, completes the protocol by spending Alice's box usually with one of his own and creating additional boxes that encode the final state of the protocol.

All the protocols here allow the first round to be offchain in the sense that Alice's box creation may be deferred until the time Bob actually participates in the protocol. Alice instead sends her box-creation transaction to Bob, who will then publish both transactions at a later time.

3.1 The XOR Game

We describe a simple game called "Same or Different" or the XOR game. Alice and Bob both submit a coin each and select a bit independently. If the bits are same, Alice gets both coins, else Bob gets both coins. The game consists of 3 steps.

1. Alice commits to a secret bit a as follows. She selects a random bit-string s and computes her commitment $k = H(s||a)$ (i.e., hash after concatenating s with a).

She creates an unspent box called the *half-game output* containing her coin and commitment k. This box is protected by a script called the *half-game script* given below. Alice waits for another player to join her game, who will do so by spending her half-game output and creating another box that satisfies the conditions given in the half-game script.

2. Bob joins Alice's game by picking a random bit b and spending Alice's half-game output to create a new box called the *full-game output*. This new box holds two coins and contains b (in the clear) alongwith Bob's public key in the registers. Note that the full-game output must satisfy the conditions given by the half-game script. In particular, one of the conditions requires that the full-game output must be protected by the *full-game script* (given below).

3. Alice opens k offchain by revealing s, a and wins if $a = b$. The winner spends the full-game output using his/her private key and providing s and a as input to the full-game script.

   If Alice fails to open k within a specified deadline then Bob automatically wins.

The full-game script encodes the following conditions: The registers R_4, R_5 and R_6 are expected to store Bob's bit b, Bob's public key (stored as a `proveDlog` proposition) and the deadline for Bob's automatic win respectively. The context variables with id 0 and 1 (provided at the time of spending the full-game box) contain s and a required to open Alice's commitnent k, which alongwith Alice's public key `alice` is used to compute `fullGameScriptHash`, the hash of the below script:

```
val s     = getVar[Coll[Byte]](0).get // bit string s
val a     = getVar[Byte](1).get       // bit a (represented as a byte)
val b     = SELF.R4[Byte].get         // bit b (represented as a byte)
val bob   = SELF.R5[SigmaProp].get    // Bob's public key
val bobDeadline = SELF.R6[Int].get
(bob && HEIGHT > bobDeadline) ||
(blake2b256(s ++ Coll(a)) == k && (alice && a == b || bob && a != b))
```

The above constants are used to create `halfGameScript` with the following code:

```
val out           = OUTPUTS(0)
val b             = out.R4[Byte].get
val bobDeadline   = out.R6[Int].get
val validBobInput = b == 0 || b == 1
validBobInput && blake2b256(out.propositionBytes) == fullGameScriptHash &&
OUTPUTS.size == 1 && bobDeadline >= HEIGHT+30 && out.value >= SELF.value * 2
```

Alice creates her half-game box protected by `halfGameScript`, which requires that the transaction spending the half-game box must generate exactly one output box with the following properties:

1. Its value must be at least twice that of the half-game box.

2. Its register R_4 must contain a byte that is either 0 or 1. This encodes Bob's choice b.

3. Its register R_6 must contain an integer that is at least 30 more than the height at which the box is generated. This will correspond to the height at which Bob automatically wins.

4. It must be protected by a script whose hash equals `fullGameScriptHash`.

The game ensure security and fairness as follows. Since Alice's choice is hidden from Bob when he creates the full-game output, he does not have any advantage in selecting b. Secondly, Alice is guaranteed to lose if she commits to a value other than 0 or 1 because she can win only if $a = b$. Thus, the rational strategy for Alice is to commit to a correct value. Finally, if Alice refuses to open her commitment, then Bob is sure to win after the deadline expires.

3.2 Rock-Paper-Scissors Game

Compared to Rock-Paper-Scissors (RPS), the XOR game is simpler (and efficient) because there is no draw condition and for this reason should be prefered in practice. However, it is useful to consider the RPS game as an example of more complex protocols.

Let $a, b \in \mathbb{Z}_3$ be the choices of Alice and Bob, with the understanding that 0, 1 and 2 represent rock, paper and scissors respectively. If $a = b$ then the game is a draw, otherwise Alice wins if $a - b \in \{1, -2\}$ else Bob wins. The game is similar to XOR, except that Bob generates two outputs to handle the draw case (where each player gets one output). Alice's commitment $k = H(a||s)$ and public key `alice` is used in generating `fullGameScriptHash`, the hash of the following script:

```
val s = getVar[Coll[Byte]](0).get  // Alice's secret byte string s
val a = getVar[Byte](1).get  // Alice's secret choice a (represented as a byte)
val b = SELF.R4[Byte].get    // Bob's public choice b (represented as a byte)
val bob = SELF.R5[SigmaProp].get
val bobDeadline = SELF.R6[Int].get // after this, it becomes Bob's coin
val drawPubKey = SELF.R7[SigmaProp].get
val valid_a = (a == 0 || a == 1 || a == 2) && blake2b256(s ++ Coll(a)) == k

(bob && HEIGHT > bobDeadline) || {valid_a &&
  if (a == b) drawPubKey else {if ((a - b) == 1 || (a - b) == -2) alice else bob}}
```

To start the game, Alice creates a box protected by the script given below:

```
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) &&
  blake2b256(out.propositionBytes) == fullGameScriptHash
} && OUTPUTS.size == 2 && OUTPUTS(0).R7[SigmaProp].get == alice
```

The above code ensures that register R_7 of the first output contains Alice's public key (for the draw scenario). Bob has to make sure 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.

3.3 ErgoMix: Non-Interactive CoinJoin

Privacy enhancing techniques in blockchains generally fall into two categories. The first is hiding the amounts being transferred, such as in Confidential Transactions [6]. The second is obscuring the input-output relationships such as in ZeroCoin [7] and CoinJoin [8]. Some solutions such as

MimbleWimble [9] and Z-Cash [10, 11] combine both approaches. We describe ErgoMix, another privacy enhancing protocol based on the latter approach. The protocol is motivated from ZeroCoin and CoinJoin to overcome some of their limitations.

ErgoMix uses a pool of *Half-Mix* boxes, which are boxes ready for mixing. This is called the *H-pool*. To mix an arbitrary box B, any one of the following is done:

1. **Pool:** Add box B to the H-pool and wait for someone to use it in a mix step.
2. **Mix:** Pick any box A from the H-pool and a secret bit b. Spend A, B to generate two *Fully Mixed* boxes O_0, O_1 such O_b and O_{1-b} are spendable by A's and B's owners respectively.

Privacy comes from the fact that boxes O_b and O_{1-b} are indistinguishable so an outsider cannot guess b with probability better than $1/2$. Thus, the probability of guessing the original box after n mixes is $1/2^n$. A box is mixed several times to reach the desired privacy. Figure 1b explains the protocol.

ErgoMix uses a primitive called a *Proof of Diffie-Hellman Tuple*, explained below. Let g, h, u, v be public group elements. The prover proves knowledge of x such that $u = g^x$ and $v = h^x$.

1. The prover picks $r \xleftarrow{R} \mathbb{Z}_q$, computes $(t_0, t_1) = (g^r, h^r)$ and sends (t_0, t_1) to the verifier.
2. The verifier picks $c \xleftarrow{R} \mathbb{Z}_q$ and sends c to prover.
3. The prover sends $z = r + cx$ to the verifier, who accepts if $g^z = t_0 \cdot u^c$ and $h^z = t_1 \cdot v^c$.

We use the non-interactive variant, where $c = H(t_0 \| t_1 \| m)$. We call this $\texttt{proveDHTuple}(g, h, u, v)$.

3.3.1 The Basic Protocol

Without loss of generality, Alice will pool and Bob will mix. Let g be the generator of `proveDlog`.

1. **Pool:** To add a coin to the H-pool, Alice picks random $x \in \mathbb{Z}_q$ and creates an output box A containing $u = g^x$ protected by the script given below. She waits for Bob to join, who will do so by spending A in a transaction satisfying following conditions:
   (a) It has two outputs O_0, O_1 containing pairs (w_0, w_1), (w_1, w_0) respectively for $w_0, w_1 \in G$.
   (b) One of $(g, u, w_0, w_1), (g, u, w_1, w_0)$ is of the form (g, g^x, g^y, g^{xy}), a valid Diffie-Hellman tuple. This is encoded as $\texttt{proveDHTuple}(g, u, w_0, w_1) \vee \texttt{proveDHTuple}(g, u, w_1, w_0)$.
   (c) The value of O_0, O_1 is the same as that of A.
   (d) Both O_0, O_1 should be protected by the script $\tau_\mathsf{A} \vee \tau_\mathsf{B}$ given in the Mix step below.
2. **Mix:** Bob picks secrets $(b, y) \in \mathbb{Z}_2 \times \mathbb{Z}_q$ and spends A with one of his own box to create two output boxes O_0, O_1 of equal value such that O_b is spendable by Alice alone and O_{1-b} by Bob alone. The boxes are indistinguisable in the sense that they have identical scripts operating on data registers c, d containing different (but related) elements from G as explained below.
   (a) Registers (c, d) of O_b and O_{1-b} are set to (g^y, u^y) and (u^y, g^y) respectively.
   (b) Each box is protected by the proposition $\tau_\mathsf{A} \vee \tau_\mathsf{B}$, where τ_A and τ_B are as follows:
   τ_A = "Prove knowledge of x such that $u = g^x$ and $d = c^x$ via $\texttt{proveDHTuple}(g, c, u, d)$."
   τ_B = "Prove knowledge of y such that $d = g^y$ via $\texttt{proveDlog}(d)$."

After the mix, Alice and Bob can spent their respective boxes using their secrets. Alice can identify her box as the one with $d = c^x$.

ErgoScript Code: First compute `fullMixScriptHash`, the hash of the following script:

```
val u = SELF.R4[GroupElement].get // copied from previous transaction
val c = SELF.R5[GroupElement].get
val d = SELF.R6[GroupElement].get
proveDlog(d) || proveDHTuple(g, c, u, d)
```

Next create a script, `halfMixScript`, having the following code:

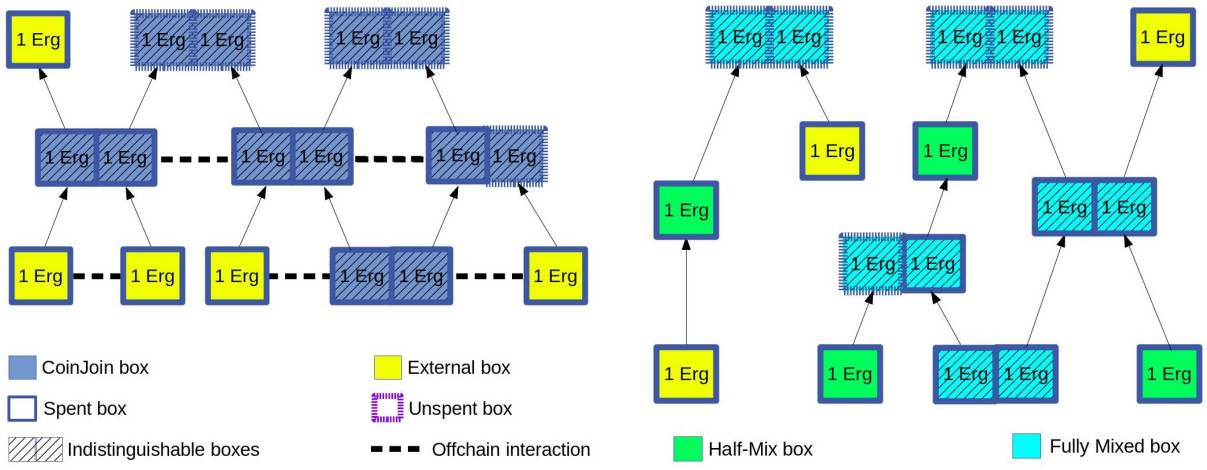
```
val u = SELF.R4[GroupElement].get
val u0 = OUTPUT(0).R4[GroupElement].get
val c0 = OUTPUT(0).R5[GroupElement].get // w0
val d0 = OUTPUT(0).R6[GroupElement].get // w1
val u1 = OUTPUT(1).R4[GroupElement].get
val c1 = OUTPUT(1).R5[GroupElement].get // w1
val d1 = OUTPUT(1).R6[GroupElement].get // w0
val bob = u0 == u && u1 == u && c0 == d1 && c1 == d0 &&
          (proveDHTuple(g, u, c0, d0) || proveDHTuple(g, u, d0, c0))
val alice = proveDlog(u)    // so Alice can spend if no one joins for a long time
val fullMixBox = {(b:Box) => blake2b256(b.propositionBytes) == fullMixScriptHash}
val fullMixTx = OUTPUT(0).value == SELF.value && OUTPUT(1).value == SELF.value &&
                fullMixBox(OUTPUT(0)) && fullMixBox(OUTPUT(1))

fullMixTx && (bob || alice)
```

Alice's Half-Mix box is protected by `halfMixScript` given above, which Bob can spend using the condition `bob`. In case no one spends her box for a long time, she can do so herself using the condition `alice`, as long as she spends it in a mix transaction.

3.3.2 Analysis Of The Protocol

Security: Observe that registers (c, d) of O_b and O_{1-b} contain (g^y, g^{xy}) and (g^{xy}, g^y) respectively, implying that O_b's spending condition reduces to `proveDlog`$(g^{xy})\vee$`proveDHTuple`(g, g^y, g^x, g^{xy}) and O_{1-b}'s reduces to `proveDlog`$(g^y) \vee$ `proveDHTuple`(g, g^{xy}, g^x, g^y). Thus, while Alice can spend O_b using `proveDHTuple` with her secret x, she cannot satisfy the spending condition of O_{1-b}. Similarly, Bob can only spend O_{1-b} using `proveDlog` with his secret y. Bob must generate O_0, O_1 this way because he must prove that one of the outputs contains a valid DH tuple.

For privacy, observe that any two identical boxes protected by `halfMixScript` have *spender indistinguisbility* because each one is spent using a Σ-OR-proof that is zero-knowledge [1]. It can be shown that any algorithm that, given (g, g^x), distinguishes (g^y, g^{xy}) from (g^{xy}, g^y) can be used to solve the *Decision Diffie Hellman* (DDH) problem. It follows that our boxes, which are of this form, are also indistingushable if the DDH problem is hard.



(a) CoinJoin (b) ErgoMix

Figure 1: Comparing CoinJoin and ErgoMix

Comparing with CoinJoin: CoinJoin [8] is a privacy enhancing protocol, where multiple parties provide inputs and create outputs in a single transaction computed interactively such that the original inputs and outputs are unlinked. The optimal use of CoinJoin is when two inputs of equal value are joined to generate two outputs of equal value, and the process is repeated, as depicted in Figure 1a. This requires two parties to interactively sign a transaction offchain and this interactive nature is the primary drawback of CoinJoin, which ErgoMix aims to overcome. In ErgoMix, this interaction is replaced by public participation using the blockchain. While this adds one more transaction, it does not require interaction between the parties. Note that ErgoMix transactions are detectable, while CoinJoin transactions are indistinguishable from ordinary transactions.

Comparing with ZeroCoin: ZeroCoin is a privacy enhancing protocol that uses a mixing pool. An ordinary coin is added to the pool as a commitment c to some secrets (r, s), and is later spent such that the link to c is not publicly visible. The value c must be permanently stored in the pool, since the spending transaction cannot reveal it. Instead, it reveals the secret s (the *serial number*) along with a zero-knowledge proof that s was used in a commitment from the pool. The serial number is permanently stored to prevent double spending. One consequence of this is that both the pool (the set of commitments) and the set of spent serial numbers must be maintained in memory for verifying every transaction. Another consequence is that the sizes of the these two sets increase monotonously. This is the main drawback of ZeroCoin (also ZCash [10]), which ErgoMix tries to address. In ErgoMix, once a box is spent, no information about it is kept in memory, and in particular no data sets of monotonously increasing sizes are maintained.

Offchain Pool: The H-Pool can be kept entirely offchain, so that Alice's Half-Mix box need not be present on the blockchain till the time Bob decides to spend it. Alice sends her unbroadcasted transaction directly to Bob who will broadcast both transactions at some later time.

Future enhancements: Compared to CoinJoin, ErgoMix requires an additional box (the Half-Mix box) as depicted in Figure 1. It will be better to have a variant that eliminates this box. One way to do this would be to find a way so that the mix step directly outputs two indistinguishable Half-Mix boxes that can be used either in the mix step or spent externally.

3.3.3 Handling Fee In ErgoMix

Similar to ZeroCoin and the canonical variant of CoinJoin in Figure 1a, each coin in ErgoMix must be of a fixed value, which is carried over to the next stage. This is fine in theory but implies zero-fee transactions, which is not possible in practice. Below we discuss some approaches for handling fee.

Assume that fee is paid in *mixing tokens*, which are tokens[1] issued by a 3rd party and that creation of a mixed output consumes one such token. A mix transaction (which has two such outputs) consumes exactly two mixing tokens and, to maintain privacy, the balance must be equally distributed between the two outputs. Below are some strategies to ensure fairness in fee payment.

1. **Perfect Fairness:** Alice's Half-Mix box contains i mixing tokens and she requires each output box to contain $i-1$ mixing tokens. Thus, Alice can mix her coin i times.

   This optimal fee strategy, however, has two drawbacks. Firstly, it has weakened privacy because it restricts the coins that can be mixed. Secondly, it impacts usability because there may not be boxes with the desired number of tokens. The approximate fairness strategy, discussed next, has better privacy and usability at the cost of reduced fairness.

2. **Approximate Fairness:** Alice relaxes her condition by requiring that Bob contribute at least one token in the mix. However, she also requires Bob to have *initially* started with exactly 1000 tokens in his first mix. Thus, if Bob is contributing, say, 1 token in the current mix, then Alice wants to ensure that he actually got there 'the hard way', by starting out with 1000 tokens and losing them in sequential mixes. This can be done as follows:

   Firstly, the token issuer must restrict the entry of tokens by issuing them only in batches of 1000 in a box protected by the script below, which requires that the tokens can be transferred (as a whole) only if the transaction is either a mix transaction or creates a Half-Mix box:

   ```
   val halfBox = {(b:Box) => blake2b256(b.propositionBytes) == halfMixScriptHash}
   val sameTokenHalfBox = {(b:Box) => halfBox(b) && b.tokens(0) == SELF.tokens(0)}
   carol && (halfBox(INPUTS(0) || sameTokenHalfBox(OUTPUTS(0))) // carol is buyer
   ```

   The value `halfMixScriptHash` is a hash of `halfMixScript`, which has the following additional code: `out.R7[Coll[Byte]].get == blake2b256(SELF.propositionBytes)`, thereby ensuring that R_7 of each output contains its hash. The code of `fullMixScript` is modified:

   ```
   val halfMixScriptHash = SELF.R7[Coll[Byte]].get
   val halfBox = {(b:Box) => blake2b256(b.propositionBytes) == halfMixScriptHash}
   val sameTokenHalfBox = {(b:Box) => halfBox(b) && b.tokens(0) == SELF.tokens(0)}
   val noToken = {(token:(Coll[Byte], Long)) => token._1 != SELF.tokens(0)._1}
   ```

[1] Every transaction may generate any quantity of at most one token, whose ID is the box-ID of the first input. For other token-IDs, the sum of quantities in outputs must be less than or equal to the sum of quantities in inputs.

```
val noTokenBox = {(b:Box) => b.tokens.forall(noToken)}
val noTokenTx = OUTPUTS.forall(noTokenBox)
(halfBox(INPUTS(0)) || sameTokenHalfBox(OUTPUTS(0)) || noTokenTx) && ...
```

3. **First Spender Pays Fee:** Another enhancement, primarily in perfect fairness, is to benefit the party that is willing to wait longer. We then require that the fee for the mix transaction be paid by the first party that spends an output. We can identify the first spender as follows.

   A mix transaction must generate exactly 4 quantities of a token (with some id x) distributed equally among 4 outputs. Two of these are the standard mix outputs O_0, O_1 with the additional spending condition that one output must contain some non-zero quantity of token x. The other two boxes, O_2, O_3, have the following identical spending conditions:

   (a) The sum of quantities of token x in the inputs and outputs is 3 and 2 respectively.

   (b) One output contains 2 quantities of token x protected by the same script as this box.

   Then it the second spender if and only if there is an input with two quantities of token x. The mix step will create an additional box with two tokens spendable by the second spender.

4 Conclusion

This article described smart contracts written in ErgoScript. The examples build upon concepts from the ErgoScript white-paper [3]. More advanced contracts will be discussed in another tutorial.

References

[1] Ivan Damgård. On Σ-Protocols, 2010. http://www.cs.au.dk/~ivan/Sigma.pdf.

[2] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[3] Ergoscript, a cryptocurrency scripting language supporting noninteractive zero-knowledge proofs. https://docs.ergoplatform.com/ErgoScript.pdf, 03 2019.

[4] Scorex Foundation. Sigmastate interpretter. https://github.com/ScorexFoundation/sigmastate-interpreter, 2017.

[5] Adding anti-theft feature to bitcoin using a new address type. https://bitcointalk.org/index.php?topic=4440801.0, 06 2018.

[6] Gregory Maxwell. Confidential transactions. https://people.xiph.org/~greg/confidential_values.txt, 2015.

[7] Ian Miers, Christina Garman, Matthew Green, and A.D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. pages 397–411, 05 2013.

[8] Coinjoin: Bitcoin privacy for the real world. https://bitcointalk.org/?topic=279249, 08 2013.

[9] T.E. Jedusor. Mimblewimble. https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt, 2016.

[10] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, Washington, DC, USA, 2014. IEEE Computer Society.

[11] Zcash. https://z.cash, 2016.