ErgoScript Overview (working title)

authors

January 22, 2019

Abstract

This paper describes uses ErgoScript to design games and mixing protocols

1 Overview of ErgoScript

Built-in Σ-protocols Our language incorporates proofs as first-class citizens, giving developers access to cryptographic primitives for non-interactive *proofs of knowledge* known as Σ-protocols (pronounced "sigma-protocols"). A transaction's output is protected by a statement known as a Σ-statement. In order to spend the output, the statement needs to be proven true (by providing a Σ-proof). The combination of the protecting script and the spending proof forms a Σ-protocol.

1.1 Sigma Protocols

For an introduction to Σ-protocols, we refer the reader to [**?**] and [**?**, Chapter 6]. Here we give a brief overview. The classic example a Σ-proof is the following 3-step identification protocol due to [**?**]. G is a cyclic multiplicative group of prime order q such that computing discrete logarithms in G is hard. Let g be a generator of G. Alice has a secret $x \in \mathbb{Z}_q$, which she wants to prove knowledge of to some Bob who knows $y = g^x$.

1. **Commit:** Alice selects a random r, computes $u = g^r$ and sends u to Bob.

2. **Challenge:** Bob selects a random $c \in \mathbb{Z}_q$ and sends c to Alice.

3. **Response:** Alice computes $z = r + cx$ and sends z to Bob. Bob accepts iff $g^z = u \cdot y^c$.

The above protocol is a proof of knowledge because Bob can extract x if he can get Alice to respond twice for the same r and different c. As an example, for $c = 1, 2$, Bob can obtain $r + x$ and $r + 2x$, the difference of which gives x. This is also called (special) soundness. Above protocol is also (honest verifier) zero-knowledge because anyone can impersonate Alice if the challenge c of Step 2 is known in advance simply by computing $z \overset{R}{\leftarrow} \mathbb{Z}_q$ and $u = g^z / y^c$.

Any protocol that has the above 3-move structure (Alice $\overset{u}{\rightarrow}$ Bob, Alice $\overset{c}{\leftarrow}$ Bob, Alice $\overset{z}{\rightarrow}$ Bob), along with zero-knowledge and soundness property for some statement s is called a Σ-protocol.

1.2 Non-Interactive Sigma Protocols

For any Σ-protocol of some statement s with messages (u, c, z), we can apply the Fiat-Shamir transform to convert it into a non-interactive proof of s by replacing the role of Bob in Step 2 by any hash function H and computing $c = H(u)$. The resulting protocol with messages $(u, H(u), z)$ can be performed by Alice alone. Intuitively, since c depends deterministically on u, Bob cannot "rewind" Alice and get two different responses for the same r. Additionally, Alice cannot know c in advance before deciding u under the random oracle assumption.

1.2.1 Digital Signatures from Sigma Protocols

Conceptually, Σ-proofs [?] are generalizations [?] of digital signatures. In fact, Schnorr signature scheme [?] (whose more recent version is popularly known as EdDSA [?, ?]) is a special case of the above identification protocol with $c = H(u\|m)$, where m is the message. The signature proves that the recipient knows the discrete logarithm of the public key (the proof is attached to a specific message, such as a particular transaction, and thus becomes a signature on the message; all Σ-proofs described here are attached to specific messages). Σ-protocols exist for proving a variety of properties and, importantly for ErgoScript, elementary Σ-protocols can be combined into more sophisticated ones using the techniques of [?].

1.3 Combining Sigma Protocols

Any two Σ-protocols of statements s_0, s_1 with messages $(u_0, c_0, z_0), (u_1, c_1, z_1)$ respectively can be combined into a Σ-protocol of $s_0 \wedge s_1$ with messages $(u, c, z) = (u_0\|u_1, c_0\|c_1, c_0\|c_1)$. We call such a construction an AND operator on the protocols.

More interestingly, as shown in [?],the two protocols can also be used to construct a Σ-protocol for $s_0 \vee s_1$, where Alice proves knowledge of the witness of one of the statements without revealing which one. Let $b \in \{0, 1\}$ be the bit such that Alice knows the witness for s_b but not for s_{1-b}. Alice will run the correct protocol for s_b and a simulation for s_{1-b}. First she generates a random challenge c_{1-b}. She then generates (u_{1-b}, z_{1-b}) by using the simulator on c_{1-b}. She also generates u_b by following the protocol correctly. The pair (u_0, u_1) is sent to Bob, who responds with a challenge c. Alice then computes $c_b = c \oplus c_{1-b}$. She computes z_b using (u_b, c_b). Her response to Bob is $((z_0, c_0), (z_1, c_1))$, who accepts if: (1) $c = c_0 \oplus c_1$ and (2) $(u_0, c_0, z_0), (u_1, c_1, z_1)$ are both accepting convesations for s_0, s_1 respectively. We call such a construction an OR operator.

Clearly, both the AND and OR operators also result in Σ-protocols that can be further combined or made non-interactive via the Fiat-Shamir transform.

There is one more operator that we need called THRESHOLD, which allows us to construct a k-out-of-n Σ-protocol in the following sense [?]: given n statements, Alice can prove knowledge of witnesses for at least k statements without revealing which statements were true.

Scalahub notes : Describe Threshold briefly

2 Primitives in ErgoScript

ErgoScript provides as primitives two elementary Σ-protocols over an elliptic curve group of prime order, written here in multiplicative notation:

1. A proof of knowledge of discrete logarithm with respect to a fixed group generator: given a group element y, the proof convinces a verifier that the prover knows x such that $y = g^x$, where g is the group generator (also known as base point), without revealing x. This is the Schnorr signature with public key y, described in Section 1.2.1. We call this primtive ProveDLog(g, y).

   Scalahub notes : What is the exact input to the hash function? (what forms the message?)

2. A proof of equality of discrete logarithms (i.e., a proof of a Diffie-Hellman tuple): given group elements g_0, y_0, g_1, y_1, the prover, Alice convinces a verifier Bob that she knows x such that $y_0 = {g_0}^x$ and $y_1 = {g_1}^x$, without revealing x. This is done as follows.

   (a) **Commit:** Alice picks $r \overset{R}{\leftarrow} \mathbb{Z}_q$, computes $(u_0, u_1) = ({g_0}^r, {g_1}^r)$ and sends (u_0, u_1) to Bob.

   (b) **Challenge:** Bob picks $c \overset{R}{\leftarrow} \mathbb{Z}_q$ and sends c to Alice.

   (c) **Response:** Alice computes $z = r + cx$ and sends z to Bob, who accepts if ${g_b}^z = u_b \cdot {y_b}^c$ for $b \in \{0, 1\}$.

   We use the non-interactive variant of this protocol, where the challenge is computed as $c = H(u_0 \| u_1)$. We call this primitive ProveDLogEq(g_0, y_0, g_1, y_1).

   Scalahub notes : What is the exact input to the hash function?

ErgoScript gives the ability to build more sophisticated Σ-protocols using the connectives AND, OR, and THRESHOLD. Crucially, the proof for an OR and a THRESHOLD connective does not reveal which of the relevant values the prover knows. For example, in ErgoScript a ring signature by public keys $y_1, \ldots, y_n$ can be specified as an OR of Σ-protocols for proving knowledge of discrete logarithms of $y_1, \ldots, y_n$. The proof can be constructed with the knowledge of just one such discrete logarithm, and does not reveal which one was used in its construction.

Rich context, enabling self-replication In addition to Σ-protocols, ErgoScript allows for predicates over the state of the blockchain and the current transaction. These predicates can be combined, via Boolean connectives, with Σ-statements, and are used during transaction validation. The set of predicates is richer than in Bitcoin, but still lean in order to allow for efficient processing even by light clients. Like in Bitcoin, we allow the use of current height of the blockchain; unlike Bitcoin, we also allow the use of information contained in the spending transaction, such as inputs it is trying to spend and outputs it is trying to create. This feature enables self-replication and sophisticated (even Turing-complete) long-term script behaviour, as described in examples below.

ErgoScript is statically typed (with compile-time type checking) and allows the usual operations, such as integer arithmetic.

Scalahub notes : This seems incomplete. In particular, we should describe all context variables and operations allowed, possibly using BNF or some grammar.

Running time estimation and safety checks Leo notes : someone should fill this in, because I know very little about it See Section ?? for more details.

3 ErgoScript Examples

We give some examples of ErgoScript to illustrate its usage.

3.1 Reversible Payments

Often lack of reversibility is considered a weak point of Bitcoin. We give a protocol that allows reversible payments in Ergo. The payments are reversible for a limited time, which we call the cooling off period. Let pk_{A} be the public key of someone who can reverse payments. This public key can (and should) be different from the sender of the payment. Let pk_{B} be the recipient of the payment. The sender will create the output with the following spending condition:

$$(pk_{\mathsf{A}} \wedge \texttt{depth <= coolingPeriod}) \vee (pk_{\mathsf{B}} \wedge \texttt{depth > coolingPeriod})$$

Scalahub notes : This requires a 'depth' instruction, which is current not present. Depth is simply current height - inclusion height.

3.2 The XOR Game

We describe a simple game called "Same or different" or the XOR game. Alice and Bob both select a secret bit and submit a coin each. They then reveal their secret bits. If the bits are same, Alice gets both coins, else Bob gets both coins. The game requires at least 3 transactions.

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

   She adds her commitment h along with her coin protected by secret x_0 and public $y_0 = g^{x_0}$. This creates an unspent Box by Alice. She waits for some Bob to join the game by spending this output subject to certian conditions given by the *firstRoundScript* (see below). Alice can spend the box if no one joins within 100 blocks.

2. Bob generates bit $b \in \{0, 1\}$ and adds it (in the clear) along with his coin protected by secret x_1 and public $y_1 = g^{x_1}$. He does this by spending Alice's box with one of his own and creating a new Box containing both coins along with Alice's commitment h and his own bit b. Lets call this an intermediate box. Note that the transaction must satisfy the conditions given by *firstRoundScript*. In particular, one of the instructions in this script requires that the output must be protected by *secondRoundScript* (also given below).

3. Alice opens h by revealing s somehow (either privately to Bob or publicly). If Alice fails to reveal a within 100 blocks of creating the intermediate box then Bob automatically wins and gets to spend the intermediate box. Anyone with knowledge of s can send the price to the winner by inputting a to *secondRoundScript*.

The script *secondRoundScript* is given by the following conditions:

1. The first free register (R4) is assumed to be Bob's input b (boolean).
2. The second free register (R5) contains Bob's public key.
3. If the block height is more than 100 then Bob's public key can spend.
4. The first typed variable (obtainable via `getVar(0)`) must be Alice's secret s.
5. The second typed variable (obtainable via `getVar(1)`) must be Alice's choice a (boolean).

6. Commitment must be correctly opened (i.e., $H(s\|a)$ must equal h).
7. If $(a == b)$ then Alice can spend else Bob can spend.

This is encoded in out Scala program as follows:

```
{
val secondRoundScript = Compile( // Can this be implemented?
"""val hash = INPUTS(0).R4 // Alice's original commitment
val bobsBit = INPUTS(0).R5 // Boolean, Bob's public bit
val aliceBytes = getVar[Array[Byte]](0) // Alice's open commitment, a 256 bit integer
val correctlyOpened = sha256(aliceBytes) == hash
val equals = bobsBit == aliceBytes(0) > 0  // use the LSB of aliceBytes
(correctlyOpened &&
(
(pkA && equals) ||
(pkB && !equals)
)
) ||
(pkB && txHeight > 100)"""
)

val pkA = // Alice's public key. Can we extract it from INPUT(0).propositionBytes ?
val pkB = // Bob's public key. Can we extract it from INPUT(1).propositionBytes ?
val b = getVar[Boolean](0) // Bob's public bit

(pkA && txHeight > 100) || {
INPUTS.size == 2 &&
INPUTS(0).value == INPUTS(1).value &&
INPUTS(0) == SELF &&
OUTPUTS.size == 1 &&
OUTPUTS(0).R4 == INPUTS(0).R4 && // Copy Alice's commitment hash to output
OUTPUTS(0).R5 == b
OUTPUTS(0).propositionBytes == outProp
}
```

Alice creates her coin with the following spending condition in the output:

```
  {
    val outProp = Compile( // Can this be implemented?
     """val hash = INPUTS(0).R4 // Alice's original commitment
        val bobsBit = INPUTS(0).R5 // Boolean, Bob's public bit
        val aliceBytes = getVar[Array[Byte]](0) // Alice's open commitment, a 256 bit integer
        val correctlyOpened = sha256(aliceBytes) == hash
        val equals = bobsBit == aliceBytes(0) > 0  // use the LSB of aliceBytes
        (correctlyOpened &&
           (
            (pkA && equals) ||
            (pkB && !equals)
          )
        ) ||
        (pkB && txHeight > 100)"""
    )

    val pkA = // Alice's public key. Can we extract it from INPUT(0).propositionBytes ?
    val pkB = // Bob's public key. Can we extract it from INPUT(1).propositionBytes ?
    val b = getVar[Boolean](0) // Bob's public bit

    (pkA && txHeight > 100) || {
      INPUTS.size == 2 &&
      INPUTS(0).value == INPUTS(1).value &&
      INPUTS(0) == SELF &&
      OUTPUTS.size == 1 &&
      OUTPUTS(0).R4 == INPUTS(0).R4 && // Copy Alice's commitment hash to output
      OUTPUTS(0).R5 == b
      OUTPUTS(0).propositionBytes == outProp
    }
```

3.3 The Mixing Protocol

We now describe a mixing protocol for Ergo called Twix, which is motivated from ZeroCash (ZC). The name Twix is a portmanteau of *Two* and *Mix*. Twix essentially mixes two coins and so provides "50% anonymity" in one mix. A coin can be successively mixed to increase the anonymity above any desired level (say 99.99999%). We do a formal analysis of the protocol later.

Twix is based on the UTXO model. Like ZC, Twix uses pooled coins for mixing. Unlike ZC, however, Twix has two UTXO pools for providing anonymity: a *half-mixed* pool (H) and a *fully-mixed* pool (F). The F pool contains coins that are anonymized. There is also the standard pool (S) that contains UTXOs for ordinary spending without any anonymity. The goal of the protocol is to move coins from O to F (possibly via H) and after sufficient mixing, back to O. The protocol contains the following transactions.

1. **Half-mix:** This takes one coin from either O or F and puts it in H.
2. **Full-mix:** This takes one coin from H, the other from either O or F and puts both coins in F such that it is impossible (for outsiders) to distinguish them.
3. **Spend:** This takes one coin from F and either puts it in O.

To participate in the mix, a user can may either add an unspent coin to the pool (which can be an output of a previous mix operation), or below operations anyone can add coins to this unspent pool. tained. Anyone can deposit ordinary currency, say Ethers (ETH) to the 2C pool and later withdraw from it at a 1:1 exchange rate. The key difference is that the size of the 2C pool depends on Twix attempts to fix some of its drawbacks discussed below. Like We describe Two-Coin, a cryptocurrency for enhancing pri- vacy of trancactions. Two-Coin (2C), like Zero-Coin (ZC) uses zero-knowledge proofs but is more storage efficient than ZC. Similar to ZC, in 2C a pool of anonymous coins is main- tained. Anyone can deposit ordinary currency, say Ethers (ETH) to the 2C pool and later withdraw from it at a 1:1 exchange rate. The key difference is that the size of the 2C pool depends on the number of unspent coins rather than number of deposited coins (as in ZC). This greatly enhances the scalability, while keeping anonymity at similar levels. In 2C, each unspent coin can be spent twice, hence the name (a withdraw is treated similar to a spend). Once a coin has been spent twice, it is removed from the pool. Thus, un- like ZC, a 2C pool does not have a monotonously increasing set. In practice, we can replace zero-knowledge proofs with zk-SNARKS at the cost of lower security.

1. ZeroCash requires a pool of mixed coins, whose size increases monotonously. Once a coin is added to the pool, it

The idea of Twix is quite general and can be implemented in other platforms. In Ergo, however, the protocol can be used without any additional platform support.

There is a pool of unmixed coins where anyone can add theirs.

The protocol is as follows:

1. **Pool:** To add a coin to the pool, Alice picks random generator $g_\mathsf{A} \in G$ and $x_\mathsf{A} \in \mathbb{Z}_q$. Let $y_\mathsf{A} = {g_\mathsf{A}}^{x_\mathsf{A}}$. Alice creates an output box A containing $(g_\mathsf{A}, y_\mathsf{A})$ and protected by the script given below. She waits for Bob to join by spending A subject to the conditions given in the script. Alice can spend A if no one joins within 100 blocks.

2. **Mix:** Bob randomly picks one unspent box from the pool, for instance, A. Bob then picks a random secret bit b and spends A with another of his own unspent box B. The spending transaction creates two new unspent boxes O_0, O_1 of equal values such that C_b is spendable only by Alice and C_{1-b} is spendable only by Bob. This is done as follows:

   (a) Bob picks secret $x_\mathsf{B} \in \mathbb{Z}_q$. Let $g_\mathsf{B} = {g_\mathsf{A}}^{x_\mathsf{B}}$ and $y_\mathsf{B} = {y_\mathsf{A}}^{x_\mathsf{B}} = {g_\mathsf{A}}^{x_\mathsf{A}x_\mathsf{B}}$ $(= {g_\mathsf{B}}^{x_\mathsf{A}})$. The box O_b contains the tuple $(g_\mathsf{A}, y_\mathsf{A}, g_\mathsf{B}, y_\mathsf{B})$ and O_{1-b} contains $(g_\mathsf{A}, y_\mathsf{A}, y_\mathsf{B}, g_\mathsf{B})$. Assuming that the *Decision Diffie-Hellman Problem* in G is hard, the distributions $(g_\mathsf{A}, {g_\mathsf{A}}^{x_\mathsf{A}}, {g_\mathsf{A}}^{x_\mathsf{B}}, {g_\mathsf{A}}^{x_\mathsf{A}x_\mathsf{B}})$ and $(g_\mathsf{A}, {g_\mathsf{A}}^{x_\mathsf{A}}, {g_\mathsf{A}}^{x_\mathsf{A}x_\mathsf{B}}, {g_\mathsf{A}}^{x_\mathsf{B}})$ are computationally indistinguishable. In other words, without knowledge of x_A or x_B, one cannot guess b with probability better than 1/2.

   (b) Let s_A be the statement: "For given $(g_\mathsf{A}, y_\mathsf{A}, g_\mathsf{B}, y_\mathsf{B})$ prove knowledge of x_A such that $y_\mathsf{A} = {g_\mathsf{A}}^{x_\mathsf{A}}$ and $y_\mathsf{B} = {g_\mathsf{B}}^{x_\mathsf{A}}$." This is encoded as

   $$s_\mathsf{A} = (g_\mathsf{A}, y_\mathsf{A}, g_\mathsf{B}, y_\mathsf{B}) \mapsto \texttt{ProveDLogEq}(g_\mathsf{A}, y_\mathsf{A}, g_\mathsf{B}, y_\mathsf{B}).$$

   (c) Let s_B be the statement: "For given $(g_\mathsf{A}, y_\mathsf{A}, y_\mathsf{B}, g_\mathsf{B})$ prove knowledge of x_B such that $g_\mathsf{B} = {g_\mathsf{A}}^{x_\mathsf{B}}$ and $y_\mathsf{B} = {y_\mathsf{A}}^{x_\mathsf{B}}$." This is encoded as

   $$s_\mathsf{B} = (g_\mathsf{A}, y_\mathsf{A}, y_\mathsf{B}, g_\mathsf{B}) \mapsto \texttt{ProveDLogEq}(g_\mathsf{A}, g_\mathsf{B}, y_\mathsf{A}, y_\mathsf{B}).$$

   Observe that the order of $g_\mathsf{B}, y_\mathsf{B}$ is reversed from s_A.

   (d) Each box is protected by the statement $s_\mathsf{A} \lor s_\mathsf{B}$.

   Alice's spending condition for A is that the transaction should be as follows:

(a) It should contain two inputs, the first of which is A. The value of the second input should be the same as in A.

(b) It should contain exactly two outputs (O_0, O_1) of the form $(g_{\mathsf{A}}, y_{\mathsf{A}}, c_0, c_1)$ and $(g_{\mathsf{A}}, y_{\mathsf{A}}, c_1, c_0)$ respectively;

(c) It The spender must satisfy $\texttt{ProveDLogEq}(g_{\mathsf{A}}, c_0, y_{\mathsf{A}}, c_1) \vee \texttt{ProveDLogEq}(g_{\mathsf{A}}, c_1, y_{\mathsf{A}}, c_0)$.

(d) The outputs should be protected by the script

3. **Spend:** Both Alice and Bob later spent their respective boxes using their secrets.