ErgoScript, a Cryptocurrency Scripting Language Supporting Noninteractive Zero-Knowledge Proofs

Ergo Developers

April 12, 2021

Abstract

This paper describes ErgoScript, a powerful and protocol-friendly scripting language for cryptocurrencies. Programs in ErgoScript are used to specify the conditions under which currency can be spent. The language supports a type of non-interactive zero-knowledge proofs called Σ-protocols and is flexible enough to allow for ring-signatures, multisignatures, multiple currencies, atomic swaps, self-replicating scripts, and long-term computation.

1 Introduction

1.1 Background

Since its early days, Bitcoin [**?**] has allowed more than simple money transfers between two public keys: its Bitcoin Script scripting language has allowed participants to specify conditions for how money could be spent. A program written in Bitcoin Script is attached to every transaction output (i.e., amount received); this program protects the transaction by determining how the transaction output can be used as an input to (i.e., spent in) a future transaction. The simplest condition is specified by a program that contains the recipient's public key and states that the money can be spent by creating a signature that verifies under this key. However, more general conditions are allowed by more sophisticated programs.

The Bitcoin Script language is a primitive stack-based language without loops [**?**]. To spend an output protected by a program, a spending transaction must provide a program in the same language, and the concatenation of the two programs must evaluate to *true*. The creator of the spending transaction can be viewed as a prover (for example, proving knowledge of the secret key by producing a signature), where the statement that needs to be proven is specified by the output that is being spent. Transaction validity is verified by evaluating programs. Bounded validation time is ensured by the absence of loops in the programming language and a maximum program size of 10 kilobytes. Even so, some denial-of-service attacks exploiting script validation time have appeared [**?**, **?**, **?**]. On the other hand, the deliberate simplicity of the programming language limits the kinds of contracts that can be created on the Bitcoin platform.

On other end of the generality spectrum, Ethereum allows for arbitrary Turing-complete programs [**?**]. This approach requires charging for computation, in order to prevent denial-of-service attacks, because the running time of a Turing-complete program cannot, in general, be estimated without actually running the program. It is also needed in this case to have gas limit per block, otherwise, DoS is possible anyway, for example, a miner can do attack the network for free.

A variety of cryptocurrency languages have appeared. We do not survey them here, but refer the reader to Scilla [?], Simplicity [?], Marlowe [?], TypeCoin [?], and many other languages.

1.2 Our Contribution: ErgoScript

In this paper we introduce a new language called ErgoScript that is specifically designed to be friendly to cryptographic protocols and applications. ErgoScript is considerably more powerful than Bitcoin Script. As ErgoScript contains no unbounded looping or recursive constructs, individual scripts in ErgoScript are not Turing-complete. In fact, given a program in ErgoScript, it is easy to obtain an estimate of its running time. However, because ErgoScript allows for self-replication, ErgoScript can be used to create Turing-complete processes in a blockchain, as shown in [?] (see also Section 2.6).

Built-in Σ-protocols Our new language incorporates proving and verifying as first-class primitives, giving developers access to a subclass of cryptographic proof systems known as non-interactive Σ-protocols (pronounced "sigma-protocols"). Thus, a script protecting a transaction output can contain statements (Σ-statements) that need to be proven (by producing Σ-proofs) in order to spend the output.

Conceptually, Σ-proofs [?] are generalizations [?] of digital signatures. In fact, Schnorr signature scheme [?] (whose more recent version is popularly known as EdDSA [?, ?]) is the canonical example of a Σ-proof: it 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 [?]. For an introduction to Σ-protocols, we refer the reader to [?] and [?, Chapter 6].

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

- A proof of knowledge of discrete logarithm with respect to a fixed group generator: given a group element h, the proof convinces a verifier that the prover knows w such that $h = g^w$, where g is the group generator (also known as base point), without revealing w. This is the same as a Schnorr signature with public key h.
- A proof that of equality of discrete logarithms (i.e., a proof of a Diffie-Hellman tuple): given group elements g_1, g_2, u_1, u_2, the proof convinces a verifier that the prover knows w such that $u_1 = g_1^w$ and $u_2 = g_2^w$, without revealing w

ErgoScript also provides the ability to build more sophisticated Σ-protocols by using connectives AND, OR, and THRESHOLD (also known as k-out-of-n). 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 $h_1, \ldots, h_n$ can be specified as an OR of Σ-protocols for proving knowledge of discrete logarithms of $h_1, \ldots, h_n$. 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.

Our implementation of these protocols is in Scala [?] and Java [?]. The implementation was informed by SCAPI [?], but does not use SCAPI code. We use Bouncy Castle [?] for big integer

and elliptic curve operations; the implementation of arithmetic in fields of characteristic 2 (for THRESHOLD connectives) is our own.

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) functional language with first-class lambda expressions, collection, tuple and optional type values, it allows standard operations, such as integer arithmetic, logical and comparison operations as well as operations on group elements and authenticated dictionaries.

2 ErgoScript Language Description

The language syntax is a subset of Scala with the same meaning, and therefore many of the constructs are easy to read for those familiar with Scala.

Before we describe the language, let us fix some terminology. A *box* (often called a "coin" in other literature) contains some amount (*value*, measured in Ergo tokens) and is protected by a *script* (boxes also contain additional information, such as other tokens; this information is described in detail in Section 2.5). A *transaction* spends the value of boxes that are its *inputs* (which are outputs of some earlier transaction) and produces boxes that are its *outputs*. In a given transaction, the sum of the values of the inputs must equal the sum of the values of the outputs (as we describe below, the scripting language is rich enough to allow for payment transactions fees and for minting of new coins without violating this rule).

All the unspent transaction outputs ($UTXO$) at a given time represent the value stored in the blockchain. A script for a box must evaluate to "true" when this box is used as an input to a transaction. This evaluation is helped by a *proof* (for Σ-statements) and a *context*, which are part of the transaction. The proof is produced by someone who knows the relevant secrets, such as secret keys; the context contains information about the spending transaction, such as details of its inputs and outputs, and the current state of the blockchain, such as the current height of the blockchain and last 10 block headers from the blockchain.

2.1 Σ-Statements

The simplest script allows the owner of a public key to spend an output box in a future transaction by issuing a signature with the corresponding secret key. If the variable `pk` holds the public key, then this script is specified simply as a string

```
pk
```

In order for the compiler to know what value the variable `pk` is referring to, the compiler needs to be supplied with an *environment*, which, in this case, is a single-element map, mapping the string

`"pk"` to the object holding the public key.[1] The value of public key is hardwired into the script at compile time, thus `"pk"` can also be called *named constant* to reflect the fact that it cannot change. When the script is later evaluated (i.e., when the box is used as a transaction input), a Σ-proof of knowledge of the corresponding secret key must be supplied by the prover.

A slightly more complex script may allow either one of two people to spend an box. If Alice owns public key `pkA` and Bob owns public key `pkB` (with corresponding secret key `skA` and `skB`), then the script that would allow either one of them to spend the box is

```
pkA || pkB
```

Again, `pkA` and `pkB` need to be mapped to public key objects by an environment map. Note that Alice and Bob individually can construct the proof necessary to spend this box using just one of the two corresponding secret keys, and, by the zero-knowledge property of Σ-protocols, the proof will not reveal whether `skA` or `skB` was used. This construct is known as a *ring* signature.

Naturally, this approach can be extended to more than two keys. If Carol's public and secret keys are `pkC` and `skC`, then a three-key ring signature can be similarly constructed as

```
pkA || pkB || pkC
```

For syntactic convenience, multiple keys can be placed into a collection, and `anyOf` operator can be used instead of `||`, as follows:

```
anyOf (Coll (pkA, pkB, pkC))
```

A conjunction is also possible: the script

```
pkA && pkB
```

requires Alice and Bob to use secret keys corresponding to `pkA` and `pkB` in order to spend the box. Note that Alice and Bob will have to communicate to produce the proof: separate signatures by Alice and Bob will not be sufficient (but Alice and Bob will not have to reveal their secret keys to each other; see Section A.2). Similarly to `||`, multiple `&&` can be used in sequence, and the operator `allOf` can be used for collections.

Using these operators, it is possible to produce more sophisticated scripts. For example, here is a script stating that the box can be spent either by Carol or by a collaboration between Alice and Bob:

```
(pkA && pkB) || pkC
```

A valid proof (contained in the spending transaction) will not reveal which of two possibilities was used.

Operators `allOf` and `anyOf` are actually special cases of the operator `atLeast(bound, collection)`, which requires the prover to know secrets for `bound` elements of the `collection` (just in like in `anyOf`, which secrets the prover used will not be revealed by the proof). `allOf(collection)` is the same as `atLeast(collection.size, collection)` and `anyOf(collection)` is the same as `atLeast(1, collection)`.

For example, here is a script stating that the box can be spent by any 3 out of the following 6 possibilities: Alice, Bob, Charlie, David and Edie, Frank and George, Helen and Irene:

[1] This map is an object in Scala, and is passed to the compiler as a parameter together with the script (which is passed in as a string) when the compiler is invoked from within Scala code. We defer the discussion of how to invoke the compiler outside of Scala code to another article.

```
atLeast(3, Coll (pkA, pkB, pkC, pkD && pkE, pkF && pkG, pkH && pkI))
```

The same result could be achieved by writing an `anyOf` of all possible 3-out-of-6 (twenty) combinations.

2.2 Mixing Σ-statements with other statements

ErgoScript allows combining statements that require proofs with other boolean statements. These statements can refer to the *context*, which has predefined variables with information about the transaction in which the script is evaluated (i.e., the box is spent). For example,

```
pkA || pkB || (pkC && HEIGHT > 500)
```

uses the predefined variable HEIGHT, which is the sequential block number (since the beginning of the blockchain, starting at 1) of the block in which the script is evaluated. This script allows Alice or Bob to spend the box before `HEIGHT` reaches 501, and Alice, Bob, or Carol to spend the box after that. If the height has not reached 501 and the box is spent, then the proof reveals that `skA` or `skB` was used in its construction, but does not reveal which one of the two. If `HEIGHT` is greater than 500, then the proof does not reveal which of the three secret keys was used. Thus, depending on the value of `HEIGHT`, the script becomes either equivalent to `pkA || pkB` or equivalent to `pkA || pkB || pkC`.

In general, script evaluation reduces the script to a Σ-statement by first evaluating all the Boolean predicates that are not Σ-statements. As we saw in the above example, the resulting Σ-statement will, in general, depend on the values of the Boolean predicates (such as whether `HEIGHT > 500`).

We emphasize that this evaluation is not the same as the usual left-to-right lazy evaluation of logical expressions, because expressions involving Σ-statements are not treated the same way as usual boolean expressions: they are evaluated last and in zero-knowledge.

In fact, `pkA` is not of type `Boolean`. It is a constant of type `SigmaProp` with the concrete value `ProveDlog(ge)`, for some public key `ge` of `GroupElement` type. The type `SigmaProp` is special in ErgoScript because it is used differently by the prover (who constructs the proof) and the verifier (who checks it). In this case `ProveDlog` require: 1) the prover (when the transaction is created) to provide a proof of knowledge of the discrete logarithm corresponing to the value `ge`; 2) the verifier (when the transaction is added to a block) to check that the proof was indeed provided.

2.3 Accessing the Context and Box Contents

In addition to the predefined variable `HEIGHT`, the context contains predefined collections `INPUTS` and `OUTPUTS`, which refer to the inputs and outputs of the spending transaction. Elements of these collections are of type `Box`. The script also has access to its own box via the context variable `SELF` of type `Box`. Note that `SELF` is also an element of the `INPUTS` collection, because the script is executed when the box is being spent.

To access information inside a box `b`, scripts can use `b.value` for the amount, `b.propositionBytes` for the protecting script, and `b.id` for the identifier of the box, which is the BLAKE2b-256 hash of the contents of the box. Boxes include additional information in *registers*; each box is unique, because one of its registers includes the transaction id in which it was created as an output, and its own index in the outputs of the transaction which created the box, accessible through `b.R3` (see Section 2.5 for more on registers).

Example: two boxes together Access to this information allows us, for example, to create an output box that can be spent only in the same transaction as another known box, and only if no other inputs are present in the transaction. If `friend` stands for an already existing box (per the environment mapping), then we create a new box that can be spent only together with `friend` and no other input by the following script (note that it uses the collection property `size` and collection indexing, starting at 0, denoted by parentheses):

```
INPUTS.size == 2 && INPUTS(0).id == friend.id
```

Note that the script does not prevent the `friend` box from being spent on its own, in which case the output box protected by the script above will become not spendable.

We can be more permissive and allow for other inputs in addition to the friend box. To do so, we will examine the input collection using the `exists` operator, which applies a boolean function to each collection element until it finds one that satisfies the function or finds that none exists. To define a function, we use lambda syntax; the argument type (in this case `Box`) is specified with a colon after the argument name `inputBox`. We name the function using the `def` keyword.

```
{
        def isFriend(inputBox: Box) = inputBox.id == friend.id
        INPUTS.exists (isFriend)
}
```

This is our first example of a script with more than one statement; note that such scripts require braces, and their output is determined by the last statement. It can also be written in one statement, as follows:

```
INPUTS.exists { (inputBox: Box) => inputBox.id == friend.id }
```

Example: crowdfunding Access to the context allows us to create a script for the following crowdfunding situation: a project backer (with key `backerPubKey`) wishes to give money to a project (with key `projectPubKey`), but only if the project raises enough money (at least `minToRaise`) from other sources by a deadline (expressed in terms of `HEIGHT`).

To give money to the project, the backer will create an output box protected by the following script. The script contains two conditions: one for the case the deadline has passed (enabling the backer to get the money back) and one for the case it succeeded (enabling the project to spend the money if the amount is at least `minToRaise` before the deadline). In order to ensure enough money has been raised, the script will search the output collection for a box with a sufficient value going to the `projectPubKey`. To check where the value of the output box is going, the script will read the script protecting the output box and compare it to the script `"projectPubKey"` (that is the simple script described in Section 2.1); bytes of this script can be obtained by `projectPubKey.propBytes`.

```
{
        val fundraisingFailure = HEIGHT >= deadline && backerPubKey
        val enoughRaised = {(outBox: Box) =>
                outBox.value >= minToRaise &&
                outBox.propositionBytes == projectPubKey.propBytes
        }
```

```
        val fundraisingSuccess = HEIGHT < deadline &&
                 projectPubKey &&
                 OUTPUTS.exists(enoughRaised)

        fundraisingFailure || fundraisingSuccess
}
```

As before, the values of `deadline`, `minToRaise`, and the two public keys are defined by the environment map and hardwired into the script at compile time.

2.4 Context Extension and Hashing

A context can also contain typed variables that can be retrieved by numerical id using the operator `getVar`. These variables are supplied by the prover specifically for a given input box (via a `ContextExtension`) together with the proof for that box. The id can be any one-byte value (from -128 to 127) and is scoped for each box separately (so variable with id 17 for one input box in a transaction is not the same as variable with id 17 for another input box in the same transaction).

Such context extensions can be useful, for example, for requiring a spending transaction to produce hash preimages (the BLAKE2b-256 and SHA-256 hash functions can be invoked in ErgoScript, using keywords `blake2b256` and `sha256`). For example,

```
pkA && blake2b256(getVar[Coll[Byte]](1).get) ==  hashOutput
```

says that spending can be done only using the signature of Alice, and only if the preimage of `hashOutput` is written in the context. Specifically, the script requires that the context extension should contain a variable (with id `1`), which is a collection of bytes that hashes to the value of `hashOutput` (the value of `hashOutput`, like `pkA`, is defined in the environment and is hardwired into the script at compile time). Note that although the script requires both the secret key corresponding to `pkA` and the hash preimage corresponding to `hashOutput`, there is the stark difference between how these two values are used: the secret key is not revealed in the proof (by the zero-knowledge property of Σ-proofs), while the hash preimage is explicitly written into the context extension and can be seen by anyone once the transaction takes place.

Example: atomic transactions and cross-chain trading Suppose there are two separate blockchains, for two different asset types. Alice wants to receive some assets in her blockchain in exchange for giving some assets to Bob in his blockchain. ErgoScript allows to accomplish it in a simpler way than proposed for Bitcoin, for example, in [**?**].

Alice creates a random secret `x` of 256 bits (32 bytes), hashes it to obtain the value `hx`, and creates a transaction in Bob's blockchain with the output box protected by the following script:

```
anyOf( Coll(
        HEIGHT > deadlineBob && pkA,
        pkB && blake2b256(getVar[Coll[Byte]](1).get) == hx
))
```

Bob can receive the value of this box only upon presentation of a hash preimage of `hx`. Alice can reclaim it after `deadlineBob`.

From this output, Bob learns `hx`. He creates a transaction in Alice's blockchain with an output box protected by the following script:

```
val x = getVar[Coll[Byte]](1).get
anyOf( Coll(
        HEIGHT > deadlineAlice && pkB,
        allOf( Coll(
                pkA,
                x.size < 33,
                blake2b256(x) == hx
        ))
))
```

If Alice is satisfied with the amount Bob is giving her, she claims the value of this box by revealing `x`. Alice is protected as long as the hash function is one-way and she keeps her `x` secret until she claims the value of this box. (She should be careful to submit her transaction in enough time before `deadlineAlice` to make sure it gets processed before Bob can reclaim this money, because once she submits the transaction, `x` is public and thus, if the `deadlineAlice` passes before the transaction is processed, Bob can both reclaim this box and claim the box Alice left in his blockchain.)

Bob is protected, because in order for Alice to claim the value of this box, she must present a hash preimage of `hx` as a context extension in the transaction that uses this box as input. But once she does, Bob also learns this hash preimage, and is able to claim the value of the box that Alice placed into his blockchain. Note that Bob needs to choose `deadlineAlice` early enough to make sure that he is able to learn the preimage of `hx` from the transaction in Alice's block chain, and create a transaction in his blockchain, all before `deadlineBob` that Alice chose. Note also that `HEIGHT` in the two scripts is with respect to two different blockchains, which may be growing at a different rate. Bob also needs to make sure that he can use Alice's `x` as a context extension; to make sure Alice cannot cheat by making this `x` so long that it will not be allowed as a context extension in his blockchain, he uses the constraint `x.size < 33`.

The same approach can be used to trade different assets on the same blockchain, in case of multi-assets blockchains. However, for transactions on a single blockchain, an alternative approach is also possible. We describe it below.

2.5 Box Registers and Additional Tokens

Together with its value and protecting script, a box can contain up to 10 numbered registers, `R0` through `R9`. The first four of these have fixed meaning, as follows. For a box `b`, `b.R0` is the same as `b.value` and `b.R1` is the same as `b.propositionBytes`.

The third register, `b.R2`, is for specifying additional, secondary tokens contained in the box (the primary token amount is specified in `b.value`). `b.R2` contains a collection of pairs, the first element of each pair specifying the token id (as a collection of 32 bytes) and the second element specifying the amount (as a long value). The maximum number of tokens in a box is set to 4. For every token id, the sum of amounts in inputs boxes must be no less than the sum of amounts in output boxes. There is one exception to this rule for the creation of new tokens. When a new token type gets created in a transaction, its id is equal to the id of the input box 0. Thus, the exception for the creation of new tokens is that if the token id in some output box is equal to the id of input box 0, then an arbitrary amount of this token can be output. Because each box has a unique id (see Section 2.3, this exception can be applied exactly once per token type. A newly created token

can be emitted in a time-controlled fashion—see Section 2.6.

The fourth register, `b.R3`, contains a pair of integer and 34-byte collection (its type then is $(Int, Coll[Byte])$). The collection specifies the 32-byte unique transaction id where this box appears as an output followed by a 2-byte sequence number of this box in the `OUTPUTS` collection of that transaction. This ensures that each box has unique `R3` and therefore a unique `id` as long as there are no hash collisions (because the `id` of the box is computed by hashing its content, including `R3`). The first element of the pair contains creation height provided by user created the box. This height could only be less or equal than real inclusion height (a transaction could not be included into the blockchain if it contains an output with creation height being no greater than current blockchain height).

The remaining six registers can be used arbitrarily.

To access a register, the type of the register needs to be specified in brackets following the register number (for example, `b.R4[Int]`). Note that `b.R4[T]` is of type `Option[T]`; `b.R4[T].isDefined` indicates whether it actually contains a value of type `T`, and `b.R4[T].get` obtains this value.

In addition to registers, scripts can access two serialized versions of the box: `b.bytes` is a serialization of the entire box including all its registers, and `b.bytesWithNoRef`, which the same but without the transaction identifier and the output index (so that a box can be viewed independently of where it appeared.

Example: atomic exchange on a single block chain These box registers provide additional capabilities to ErgoScript. Consider, for example, Alice and Bob who want to exchange tokens: they agree that Bob will give Alice 60 tokens of type `token1` (this type is mapped to an actual token id in the environment map) in exchange for 100 Ergo tokens. Alice could create an output box with value 100 and protect it with the following script:

```
(HEIGHT > deadline && pkA) || {
  val tokenData = OUTPUTS(0).R2[Coll[(Coll[Byte], Long)]].get(0)
  allOf(Coll(
      tokenData._1 == token1,
      tokenData._2 >= 60L,
      OUTPUTS(0).propositionBytes == pkA.propBytes,
      OUTPUTS(0).R4[Coll[Byte]].get == SELF.id
  ))
}
```

This script ensures that the box can be spent only in a transaction that produces an output with 60 tokens of type `token1` and gives them to Alice (Alice can reclaim the box after the deadline). Moreover, the last condition (`OUTPUTS(0).R4[Col[Byte]].get == SELF.id`) ensures that if Alice has multiple such boxes outstanding at a given time, each will produce a separate output that identifies the corresponding input. This condition prevents the following attack: if Alice has two such boxes outstanding but the last condition is not present, then they can be both used in a single transaction that contains just one output with 60 tokens of type `token1` — the script of each input box will be individually satisfied, but Alice will get less only half of what owed to her.

Bob, similarly, could create an output box with value about 0 and 60 tokens of type `token1` and protect it by the following script:

```
(HEIGHT > deadline && pkB) ||
allOf( Coll(
        OUTPUTS(1).value >= 100L,
        OUTPUTS(1).propositionBytes == pkB.propBytes,
        OUTPUTS(1).R4[Coll[Byte]].get == SELF.id,
 ))
```

A transaction containing these two boxes as inputs must produce two outputs: the first giving at least 60 tokens of type1 to Alice and the second giving at least 100 tokens of type2 to Bob. Once the two boxes are on the blockchain, anyone can create such a transaction using the two boxes as inputs, and thus effect the exchange between Alice and Bob. Unlike the cross-chain trading example above using hashing (which requires one side to go first), there are no potential problems with synchronization here, because the exchange will happen in a single transaction or will not happen at all.

2.6 Self-Replicating Code

Access to box registers allow us to create self-replicating boxes, because a script can check that an output box contains the same script as `SELF`. As shown in [**?**], this powerful tool allows for Turing-completeness as computation evolves from box to box, even if each individual script is not Turing-complete. We will demonstrate two examples of complex behavior via self-replication.

Example: time-controlled coin emission In this example, we will create a self-replicating box that emits new coins at each time step in order to add to the total amount of currency available. This box appears as an output in the genesis block (at height 0) of the blockchain; all "new" coins come from this box or its descendants, thus maintaining the invariant that for every transaction after the genesis block, the combined value of all inputs is equal to the combined value of all outputs.

The value of this box initially is equal to the total amount of currency that will eventually be available. This value will go down by a prespecified amount each time this box is transacted. Because in each transaction, the sum of input values must equal the sum of output values, when the value of this box goes down, the difference must be claimed by someone. The box is set up to allow the difference to go to anyone—presumably, it will be claimed by the miner who created the block that contains the transaction. This box will store, in `R4`, the height at which it was created. Using this information, it will be able to determine how much value to emit. The box will be set to emit a fixed amount specified by `fixedRate` per block until `HEIGHT` reaches `fixedRatePeriod`, and a linearly decreasing amount thereafter. The script will verify that the output box has the same script as itself, and that the new height stored in `R4` and the new value are correctly computed (and that the height has increased, so that a miner cannot emit more than once per block). The following script accomplishes this goal:

```
{
        val epoch = 1 + ((HEIGHT - fixedRatePeriod) / epochLength)
        val out = OUTPUTS(0)
        val coinsToIssue = if(HEIGHT < fixedRatePeriod) fixedRate
                           else fixedRate - (oneEpochReduction * epoch)
        val correctCoinsConsumed = coinsToIssue == (SELF.value - out.value)
        val sameScriptRule = SELF.propositionBytes == out.propositionBytes
```

```
        val heightIncreased = HEIGHT > SELF.R4[Int].get
        val heightCorrect = out.R4[Int].get == HEIGHT
        val lastCoins = SELF.value <= oneEpochReduction
        allOf(Coll(
                heightCorrect,
                heightIncreased,
                sameScriptRule,
                correctCoinsConsumed))
        || (heightIncreased && lastCoins)
}
```

Example: arbitrary computation via a simple cellular automaton The example in the paragraph is not meant for practical implementation; rather, it is here merely to demonstrate the Turing-complete power of self-replication. It implements the so-called "rule 110" one-dimensional cellular automaton [**?**], which is known to be Turing-complete [**?**] (with only polynomial-time overheard — i.e., P-complete [**?**]). See [**?**] for more details. The code for this example is too complex to be put here; it is available at `https://github.com/ScorexFoundation/sigmastate-interpreter/blob/master/src/test/scala/sigmastate/utxo/examples/Rule110Specification.scala`.

3 Implementation

A script is converted to an abstract syntax tree using standard compilation techniques. At evaluation time, values from the context are substituted for relevant variables and the tree is evaluated. There are two significant differences from standard compilation and evaluation techniques:

- We estimate the time required to process the script and, if it exceeds a certain bound, refuses to evaluate it in order to prevent a denial-of-service attack.
- The evaluation converts the script not to a Boolean value, but to a Σ-statement. This statement is a tree, with `proveDlog` or `proveDHtuple` nodes for leaves, and AND (`&&`), OR (`||`), or THRESHOLD for non-leaves. The prover (when trying to use in a transaction the box that is protected by the script) generates a proof this Σ-statement. The verifier verifies it, obtaining a Boolean value.

We describe the latter non-standard step in Appendix A, while the former will be described in a separate upcoming document.

4 Further Work

The next steps we plan to do (and have partially done) after releasing this document are:

1. More examples, including non-interactive and fully on-chain tumbler for mixing the coins, cold wallets, oracles, initial coin offering scenario, multi-state contract defined as a finite state machine, and so on. We already have code for such the examples done, and writing documentation about them at the moment.
2. Detailed description of used type system, cost estimation procedure, safety guarantees, and abstract syntax tree format.

References

A Implementation of Noninteractive Σ-protocols for an arbitrary And/Or/Threshold composition

A.1 Background

In this section, we explain in detail how the Σ-protocol proving and verifying is implemented. Consider the tree after the evaluation process, as described in Section 3, reduces it to only Σ-protocol nodes. Then the leaves of the tree are atomic Σ-protocols, and non-leaves are of one of three types: AND (corresponding to `&&` or `allOf` in ErgoScript), OR (corresponding to `||` or `anyOf` in ErgoScript), or THRESHOLD(k) (corresponding to `atLeast` in ErgoScript).

The meaning of the proof corresponding to THRESHOLD(k) is "the prover knows witnesses for at least k children of this node". Semantically, AND and OR are simply special cases of THRESHOLD: the meaning of the proof corresponding AND (respectively, OR) is "the prover knows witnesses for all children (respectively, at least one child) of this node". However, AND and OR are implemented differently from THRESHOLD for efficiency.

For the purposes of this description, it does not matter what specific atomic Σ-protocols are used at the leaves. They can be, for example, `proveDlog(x)` for proving knowledge of the discrete logarithm w of $x = g^w$, or `proveDHtuple(g1, g2, u1, u2)` for proving that (g_1, g_2, u_1, u_2) form a Diffie-Hellman tuple, i.e., $\exists w$ such that $u_1 = g_1^w \wedge u_2 = g_2^w$. In general, we will assume the prover has some secret w and wants to prove some property of it.

A Σ-protocol consist of three messages:

- a *commitment* a sent from the Prover to the Verifier (computed using the witness w and some freshly generated secret randomness r by the Prover's first step algorithm);
- a uniformly random *challenge* e sent from the Verifier to the Prover;
- and a *response* z sent from the Prover to the Verifier (computed using the randomness r, the challenge e, and the witness w by the Prover's second step algorithm).

The verifier then checks that the triple (a, e, z) satisfies some formula and, if so, accepts the proof.

In order to make an atomic Σ-protocol non-interactive using the so-called Fiat-Shamir heuristic, the Prover would compute e by hashing a, and the Verifier would check that e is indeed a hash of a. However, once atomic Σ-protocols are composed using AND, OR, and THRESHOLD, the challenge computation becomes more involved, as we describe below.

Σ-protocols have the property of *special honest-verifier zero-knowledge*. For the purposes of this description, it means that given a random e, it is possible to compute a and z without knowing the secret witness w that is normally needed by the prover. This computation is called "simulation". Moreover, the triple (a, e, z) computed via simulation is distributed identically to the triple (a, e, z) that results form a Σ-protocol that is run by the honest prover (who knows the secret witness w) and verifier (who generates a uniform e). The trick that makes simulation possible is that the response z is chosen by the simulator before commitment a, in contrast to the prover, who is forced to choose a before z.

Σ-protocols used in this work must also satisfy a property of *special soundness*, which means that given two triples (a_1, e_1, z_1) and (a_2, e_2, z_2) that are both accepted by the verifier, and $a_1 = a_2$

while $e_1 \neq e_2$, it is possible to compute the Prover's secret witness w in polynomial time and thus directly verify that the statement claimed by the Prover is true. Note that this computation is never performed, because the Prover will never actually answer two different challenges $e_1 \neq e_2$ for the same commitment $a_1 = a_2$.

In order for composition of Σ-protocols using AND, OR, and THRESHOLD to work, the challenge e in all protocols must be a binary string of the same length, which we will call t. (This implies that if the Verifier performs arithmetic modulo q on e, then $2^t < q$; else it would be trivial to have $e_1 \neq e_2$ by letting $e_2 = e_1 + q$, and the same z would work for both e_1 and e_2, violating the special soundness property mentioned above, because such a, e_1, z can be computed without knowledge of the witness w by using the simulator.) Note that t is the *soundness parameter*: the probability that a malicious Prover can fool the Verifier in a single attempt is 2^{-t}.

For ease of description, we will also impose the following limitation on the Σ-protocols: in the interactive version, the verification code must proceed by computing a' from z and e, and then checking if $a' = a$. That means that in the noninteractive version, the Prover needs to transmit only z and e (omitting a) to the Verifier, who will compute a' from z and e and then check that e was computed correctly as a function of a'. If the hashing is second-preimage-resistant, an incorrect a' will lead to a mismatch of e, which will be detected by the Verifier. This limitation is satisfied by most common Σ-protocols; it is not essential and can be removed by having the Prover additionally send a.

We show the steps of the prover and verifier given such a tree. While AND, OR, and THRESHOLD composition of Σ-protocols has been addressed in the literature before [**?**], prover and verifier algorithms have been described only for a single node, in terms of prover, verifier, and simulator algorithms for its children. Recursively constructing code for an entire tree by dynamically constructing prover, verifier, and simulator code for each node is inefficient. Here we take a different approach by explicitly describing how and in what order the protocol messages for each node of the tree need to be computed and verified. We are not aware of any similar descriptions in the literature.

Our description uses two kinds of tree traversal: bottom-up (also known as post-order) and top-down (also known as preorder). In a bottom-up traversal, for each node, operations are recursively applied to the children of the node before being applied to the node itself. In a top-down traversal, for each node, operations are applied to the node itself before being recursively applied to each of its children.

A.2 Proving

The Prover will know secret witnesses w for some of the leaves; for such leaves, it will choose whether to produce real or simulated proofs according to the algorithm described in this section. The Prover will simulate proofs for all other leaves.

Multiple provers There may be situations when the secret witnesses w at the leaves are distributed among several different provers, who will have to cooperate but will not have to reveal the secrets to each other; in this case, we assume there is a main prover carrying out the steps below, and we explain what the other provers need to do in order for the main prover to construct the proof. We caution that the proofs will not be zero-knowledge to the other provers nor to an adversary who can observe the communication between the provers—even just the existence of communication will reveal which parts of the tree are real and which are simulated. Moreover,

if the main prover is malicious, it may be able to attack the other provers — see Step 9 below. More sophisticated multi-party computation protocols to address this problem are available, but are outside of the scope of this paper.

Side-channel attacks Note that the algorithm described below does not attempt to be secure against side channel attacks, such as, for example, timing attacks. In particular, it may take different amounts of time depending on which nodes are real and which are simulated; a timing attacker may therefore obtain information about the set of leaves for which the Prover knows the secrets. If timing attacks are a concern, it is not enough to make sure that simulation and proving take a similar amount of time for each atomic Σ-protocol used in the leaves; implementations should also make sure that tree traversals take the same amount of time regardless of where simulated and real Σ protocols are located in the tree. In particular, implementations should avoid the use of lazy-evaluation constructs, such as "forall" and "exists".

The steps For each node in the tree, the prover will maintain a flag indicating whether it is real or simulated. Each node will also have a t-bit challenge. The leaves will additionally have two protocol values: a challenge and a response. The pseudocode below explains how these values are computed. In the pseudocode below, the word "random" should be read to also allow securely generated pseudorandom values.

The prover first has to decide which nodes will have real proofs (for which witnesses are required) and which will be simulated. For example, in an OR proof, only one of the children will be real. The prover will do so based on witnesses that are available, in three steps:

1. This step will mark as "real" every node for which the prover can produce a real proof. This step may mark as "real" more nodes than necessary if the prover has more than the minimal necessary number of witnesses (for example, more than one child of an OR). This will be corrected in the next step. In a bottom-up traversal of the tree, do the following for each node:
    - If the node is a leaf, mark it "real" if the witness for it is available; else mark it "simulated"
    - If the node is OR, mark it "real" if at least one child is marked real; else mark it "simulated"
    - If the node is AND, mark it "real" if all of its children are marked real; else mark it "simulated"
    - If the node is THRESHOLD(k), mark it "real" if at least k of its children are marked real; else mark it "simulated"
2. If the root of the tree is marked "simulated" then the prover does not have enough witnesses to perform the proof. Abort.
3. This step will change some "real" nodes to "simulated" to make sure each node has the right number of simulated children. In a top-down traversal of the tree, do the following for each node:

- If the node is OR marked "real", mark all but one of its children "simulated" (the node is guaranteed by step 1 to have at least one "real" child). Which particular child is left "real" is not important for security; the choice can be guided by efficiency or convenience considerations.
- If the node is THRESHOLD(k) marked "real", mark all but k of its children "simulated" (the node is guaranteed, by the previous step, to have at least k "real" children). Which particular ones are left "real" is not important for security; the choice can be guided by efficiency or convenience considerations.
- If the node is marked "simulated", mark all of its children "simulated"

Now the prover will compute protocol values for every node, as follows:

4. In a top-down traversal of the tree, compute the challenges e for simulated children of every node, as follows:
   - If the node is marked "real", then each of its simulated children gets a fresh uniformly random challenge in $\{0,1\}^t$. (Note that a real AND node has no simulated children, so this step applies only to real OR and THRESHOLD nodes.)
   - If the node is marked "simulated", let e_0 be the challenge computed for it. All of its children are simulated, and thus we compute challenges for all of them, as follows:
     - If the node is AND, then all of its children get e_0 as the challenge
     - If the node is OR, then each of its children except one gets a fresh uniformly random challenge in $\{0,1\}^t$. The remaining child gets a challenge computed as an XOR of the challenges of all the other children and e_0.
     - If the node is THRESHOLD(k), assume it has n children numbered from 1 to n. There are two possible algorithms. The first algorithm is faster than the second. However, it is also faster than the algorithm for a "real" THRESHOLD node; therefore, if a timing attack on the prover is a possibility, then it should not be used, because the timing attack may be able to distinguish a "real" THRESHOLD node from a "simulated" one.
       (a) The faster algorithm is as follows. Pick $n-k$ fresh uniformly random values $q_1, \ldots, q_{n-k}$ from $\{0,1\}^t$ and let $q_0 = e_0$. Viewing $1, 2, \ldots, n$ and $q_0, \ldots, q_{n-k}$ as elements of GF(2^t), evaluate the polynomial $Q(x) = \sum q_i x^i$ over GF(2^t) at points $1, 2, \ldots, n$ to get challenges for child $1, 2, \ldots, n$, respectively.
       (b) The algorithm with better resistance to timing attacks is as follows. Pick $n-k$ fresh uniformly random values $e_1, \ldots, e_{n-k}$ as challenges for the children number $1, \ldots, n-k$. Let $i_0 = 0$. Viewing $0, 1, 2, \ldots, n$ and $e_0, \ldots, e_{n-k}$ as elements of GF(2^t), find (via polynomial interpolation) the lowest-degree polynomial $Q(x) = \sum_{i=0}^{n-k} a_i x^i$ over GF(2^t) that is equal to e_j at j for each j from 0 to $n-k$ (this polynomial will have $n-k+1$ coefficients, and the lowest coefficient will be e_0). Set the challenge at child j for $n-k < j \leq n$ to equal $Q(j)$.
5. For every leaf marked "simulated", use the simulator of the Σ-protocol for that leaf to compute the commitment a and the response z, given the challenge e that is already stored in the leaf.

6. For every leaf marked "real", use the first prover step of the Σ-protocol for that leaf to compute the necessary randomness r and the commitment a. In case of multiple provers responsible for different leaves, each prover individually computes the randomness r for that leaf; the provers send their commitments a to the main prover (note that the existence of this communication, if observed the adversary, will reveal to the adversary which leaves are real).

7. Convert the tree to a string s for input to the Fiat-Shamir hash function. The conversion should be such that the tree can be unambiguously parsed and restored given the string. For each non-leaf node, the string should contain its type (AND, OR, or THRESHOLD(k)). For each leaf node, the string should contain the Σ-protocol statement being proven and the commitment. The string should not contain information on whether a node is marked "real" or "simulated", and should not contain challenges, responses, or the real/simulated flag for any node.

8. Compute the challenge for the root of the tree as the Fiat-Shamir hash of s (and, if applicable, the associated data, such as the message being signed).

9. Perform a top-down traversal of only the portion of the tree marked "real" in order to compute the challenge e for every node marked "real" below the root and, additionally, the response z for every leaf marked "real" (note that nodes marked "simulated" have all their descendants marked "simulated" and all the challenges for these descendants already computed, so there is no need to recurse down "simulated" nodes, unless timing attacks are a concern). For every node, do the following:

   - If the node is a non-leaf marked "real" whose challenge is e_0, proceed as follows:
     - If the node is AND, let each of its children have the challenge e_0
     - If the node is OR, it has only one child marked "real". Let this child have the challenge equal to the XOR of the challenges of all the other children and e_0
     - If the node is THRESHOLD(k), number its children from 1 to n. Let $i_1, \ldots, i_{n-k}$ be the indices of the children marked "simulated" and $e_1, \ldots, e_{n-k}$ be their corresponding challenges. Let $i_0 = 0$. Viewing $0, 1, 2, \ldots, n$ and $e_0, \ldots, e_{n-k}$ as elements of GF(2^t), find (via polynomial interpolation) the lowest-degree polynomial $Q(x) = \sum_{i=0}^{n-k} a_i x^i$ over GF(2^t) that is equal to e_j at i_j for each j from 0 to $n-k$ (this polynomial will have $n-k+1$ coefficients, and the lowest coefficient will be e_0). For child number i of the node, if the child is marked "real", compute its challenge as $Q(i)$ (if the child is marked "simulated", its challenge is already $Q(i)$, by construction of Q).
   - If the node is a leaf marked "real", compute its response z according to the second prover step of the Σ-protocol given the randomness r used for the commitment a, the challenge e, and witness w. In case of multiple provers, the main prover will send the relevant challenges to other provers, who will respond with the relevant z. It is crucial for security that the provers do not respond to more than one challenge e for a given commitment a. Again, like in Step 6, the mere existence of this communication reveals which leaves are real and which are simulated. Moreover, if the main prover chooses the challenge e maliciously for another prover, the zero-knowledge property of the other prover's secret is not preserved (although, for most Σ-protocols, no useful information

about the other prover's secret will be revealed for any, even maliciously chosen, e; but a malicious choice of e may make side-channel attacks easier).

10. Output the proof consisting of the following information:

    - The challenge of the root node
    - For every OR node: the challenges of all of its children but the rightmost
    - For every THRESHOLD node: the coefficients $q_1 \dots q_{n-k}$ of the polynomial $Q(x)$ (excluding a_0)
    - The responses for every leaf node

A.3 Verifying

For each node in the tree, the verifier will obtain a t-bit challenge e by either reading it directly from the proof or by using other information provided in the proof. For the leaves, the verifier will also read the response z from the proof and will re-compute, using the challenge and the response, the commitment a. Finally, verifier will hash the whole tree and will check that the hash value matches the challenge of the root node.

Note that, unlike the prover, the verifier has no way of knowing which nodes are real and which are simulated. In fact, making sure that the verifier cannot tell the difference between real and simulated nodes is one of the security goals of the protocol.

The pseudocode below details the verifier's steps.

1. Read the root challenge from in the proof.

2. In a top-down traversal of the tree, obtain the challenges for the children of every non-leaf node by reading them from the proof or computing them, as follows. Let e_0 be the challenge in the node.

    - If the node is AND, then all of its children get e_0 as the challenge
    - If the node is OR, then each of its children except rightmost reads its challenge from the proof. The rightmost child gets a challenge computed as an XOR of the challenges of all the other children and e_0.
    - If the node is THRESHOLD(k), let the number of its children be n. Assume the children are numbered from 1 to n. Let $q_0 = e_0$ and read the values $q_1, \dots, q_{n-k}$ from the proof. Viewing $1, 2, \dots, n$ and $q_0, \dots, q_{n-k}$ as elements of $\mathrm{GF}(2^t)$, evaluate the polynomial $Q(x) = \sum q_i x^i$ over $\mathrm{GF}(2^t)$ at points $1, 2, \dots, n$ to get challenges for child $1, 2, \dots, n$, respectively.

3. For every leaf node, read the response z provided in the proof.

4. For every leaf node, compute the commitment a from the challenge e and response z, per the verifier algorithm of the leaf's Σ-protocol. If the verifier algorithm of the Σ-protocol for any of the leaves rejects, then reject the entire proof.

5. Convert the tree to a string s for input to the Fiat-Shamir hash function, using the same conversion as the prover in Step 7

6. Accept the proof if the challenge at the root of the tree is equal to the Fiat-Shamir hash of s (and, if applicable, the associated data). Reject otherwise.