A Formal Specification of a Multi-Signature Scheme using Scripts

Jared Corduan
jared.corduan@iohk.io

Matthias Güdemann
matthias.gudemann@iohk.io

May 29, 2019

Abstract

This document specifies a way to support multi-signature transactions. It is based on a simple script model which uses only single step script execution and does not require data scripts.

A multi-signature scheme allows an unspent transaction output to be used as an input to a new transaction if a pre-defined combination of signatures is provided, e.g., two persons have to sign simultaneously, two out of three keys have to be provided, etc.

For an output that is locked with a multi-signature, the set of all keys that signed the transaction is given to the script for validation. In this way, the script can decide in a single step whether it has the correct key combination to spend the output.

1 Introduction

This specification for a simple multi-signature scheme is based on Corduan et al. (2019) which formally specifies the Shelley Cardano ledger. The main changes are the following:

- Add a new address type that for outputs locked by scripts.
- Add a new witness type to the transaction.
- Adapt the transaction validation in such a way that funds locked by a multi-signature script can be spent.
- Adapt the functions used in the validation for the extended types of addresses and transaction inputs with scripts.

In this approach for multi-signature, the scripts receive the set of keys which were used to sign the transaction. The script can then check those against its own representation of which keys in which combination can unlock the unspent output.

This means that the scripts are completely stateless and no data needs to be supplied to the script apart from the information about which keys and in which combination can sign the transaction. It allows for any type of \( n \) out of \( m \) required signatures for a specific unspent transaction output.

2 Types

In Figure 1 the Addr type of Corduan et al. (2019) is changed to include both public key and script addresses, split into the sub-types Addr\textsubscript{vkey} and Addr\textsubscript{script}. The new script addresses contain the hash of the validator script. In accordance with the extended UTxO specification of Plutus Team...
Abstract types

\( \text{script} \in \text{Script} \)  
script type

Derived types

\( \text{addr}_s \in \text{Addr}^{\text{script}} = \text{Addr}^{\text{script}}_{\text{base}} \cup \text{Addr}^{\text{script}}_{\text{enterprise}} \cup \text{Addr}^{\text{script}}_{\text{ptr}} \)  
Script address

\( \text{addr}_{vk} \in \text{Addr}^{vkey} = \text{Addr}^{vkey}_{\text{base}} \cup \text{Addr}^{vkey}_{\text{enterprise}} \cup \text{Addr}^{vkey}_{\text{ptr}} \)  
VKey address

\( \text{addr} \in \text{Addr} = \text{Addr}^{\text{script}} \cup \text{Addr}^{vkey} \)  
Script address

Accessor Functions

\( \text{paymentHK} \in \text{Addr}^{vkey} \rightarrow \text{HashKey}_{\text{pay}} \)  
hash of payment key from addr

\( \text{validatorHash} \in \text{Addr}^{\text{script}} \rightarrow \text{Hash}_{\text{script}} \)  
hash of validator script

\( \text{stakeHK}_b \in (\text{Addr}^{vkey} \cup \text{Addr}^{\text{script}}) \rightarrow \text{HashKey}_{\text{stake}} \)  
hash of stake key for base addresses

\( \text{addrPtr} \in (\text{Addr}^{vkey}_{\text{ptr}} \cup \text{Addr}^{\text{script}}_{\text{ptr}}) \rightarrow \text{Ptr} \)  
pointer from pointer addresses

Abstract Functions

\( \text{hashScript} \in \text{Script} \rightarrow \text{Hash}_{\text{script}} \)  
hash a serialized script

Figure 1: Types for Scripts and Script Addresses

(2019), the producer signs the script when creating and the consumer provides it later when spending the output.

A transaction output that is locked by a script carries the hash of the validator script. The output can only be spent if the matching script is presented and validates its input. The \( \text{Addr}^{\text{script}} \) sub-type of \( \text{Addr} \) carries the necessary information and can therefore be part of a transaction output that consists of a pair of \( \text{Addr} \times \text{Coin} \). Analogously to \( \text{Addr}^{vkey} \), \( \text{Addr}^{\text{script}} \) also has an \textit{enterprise} script address sub-type which does not allow for using the locked funds in staking, as well as \textit{base} and \textit{pointer} script address sub-types which allow for staking in the same way as \( \text{Addr}^{vkey}_{\text{base}} \) and \( \text{Addr}^{vkey}_{\text{ptr}} \).

The hashScript function calculates the hash of a script by serializing and then hashing it. The accessor function validatorHash returns the hash of a script of a script address. The domain of the accessor function paymentHK is changed to pubkey addresses. The domains of the accessor functions stakeHK\(_b\) and addrPtr are extended to also include the respective script address variants.

In Figure 2 the type of a transaction from Corduan et al. (2019) is extended to carry an additional witness type. This is achieved by explicitly defining TxWitness as a type of the form of a pair of pubkey witnesses and script witnesses. The former accessor function txwits is renamed to txwitsVKey. The new accessor function txwitsScript returns a map of script hashes to validator scripts of a transaction. All scripts in the map need to validate the transaction in order for it to be accepted.

PendingTx is a representation of the pending transaction. In particular, this information contains the set of keys that signed the transaction. The function txPending constructs the necessary information about a transaction which can be passed as value of type PendingTx to the validator script.

In order to spend funds locked by a multi-signature script, the validator scripts need to validate the transaction. The abstract function validateScript corresponds to such a validator script. Its type consists of two parameters of unit type and one parameter of type PendingTx; its
Abstract Type

\[ \text{pendingTx} \in \text{PendingTx} \quad \text{information about pending Tx} \]

Transaction Type

\[ \begin{align*}
\text{wit} & \in \text{TxWitness} \quad = \quad (\text{VKey} \mapsto \text{Sig}, \text{Hash}_{\text{script}} \mapsto \text{Script}) \\
\text{tx} & \in \text{Tx} \quad = \quad \text{TxBody} \times \text{TxWitness} \times \text{UpdatePayload}
\end{align*} \]

Accessor Functions

\[ \begin{align*}
\text{txwitsVKey} & \in \text{Tx} \rightarrow (\text{VKey} \mapsto \text{Sig}) \quad \text{VKey witnesses} \\
\text{txwitsScripts} & \in \text{Tx} \rightarrow (\text{Hash}_{\text{script}} \mapsto \text{Script}) \quad \text{script witnesses}
\end{align*} \]

Abstract Functions

\[ \begin{align*}
\text{txPending} & \in \text{Tx} \rightarrow \text{PendingTx} \quad \text{Get necessary information from Tx} \\
\text{validateScript} & \in (\cdot) \rightarrow (\cdot) \rightarrow \text{PendingTx} \rightarrow \mathbb{B} \quad \text{validator script}
\end{align*} \]

Figure 2: Types for Transaction Inputs with Scripts

The return type is Boolean. The first two input parameters correspond to the redeemer and the data scripts which are used in the full extended UTxO model for Plutus Team (2019). As those values are not required for simple multi-signature, we use the unit type for them. The Boolean return type signals whether the script succeeded in validating the transaction.

The following is a possible Plutus implementation of a simple \( n \) out of \( m \) multi-signature validation script. The type \( \text{MultiSig} \) is a list of keys and a threshold value.

```haskell
import qualified Language.PlutusTx as P
import Ledger.Validation as V

data MultiSig = MultiSig
  { signatories :: [Ledger.PubKey]
  , requiredSignatures :: Integer
  } -- ^ List of public keys of people who may sign the transaction

-- ^ Minimum number of signatures required to unlock
-- the output (should not exceed \( \text{length signatories} \))

validate :: MultiSig -> () -> () -> PendingTx -> Bool
validate multiSig@(MultiSig keys num) () () p =
  let present = P.length (P.filter (V.txSignedBy p) keys)
  in present `P.geq` num
```

The above Plutus script takes a parameter \( \text{multiSig} \) of type \( \text{MultiSig} \) which is a list of keys \( \text{keys} \) and a threshold \( \text{num} \) which indicates how many of the keys are required as signatures. When the validation script is called, it computes the list of keys in \( \text{keys} \) which signed the transaction. If the length of that list is greater than or equal to \( \text{num} \), then enough signatures for the multi-signature are present. Therefore, with \( \text{multiSig} \) being a value of type \( \text{MultiSig} \), calling \( \text{validate multiSig} \) returns a validator script with the correct type according to Figure 2.

Figure 3 shows the helper functions \( \text{txInsVKey} \) and \( \text{txInsScript} \) which partition the set of transaction inputs of the transaction into those that are locked with a private key and those that are locked via a script. The helper function \( \text{validators} \) constructs a map from transaction inputs
to scripts where for each input, the corresponding output of the UTxO can only be spent if the script validates the transaction.

### 3 Ledger Transition for Multi-Signature

The main change for the ledger transitions when using script based multi-signature is the validation of the UTxOW transition of Corduan et al. (2019). Its extended transition system is shown in Figure 4, in the format described in Formal Methods Team (2018). The constraint on the set of required witnesses is relaxed in such a way that “redundant” signatures can be supplied in the transaction. The complete set of verification keys is then passed to the validator script as part of pendingTx when validating all supplied scripts.

The set of all validator scripts of \( \text{txinsScript} \) \((\text{txins tx})\) is checked for:

- the script hash being equal to the hash stored in the output to spent (done in the function \( \text{validators} \)) and
- the validator script validating the transaction.

We also check that the size of the set of spent outputs locked by a script is equal to the elements of the inputs for which we have a validator script. Overall that means that for each spent output we have a signature or a validation script.

### 4 Alternative Implementation

An alternative implementation for multi-signature scripts is an embedding of the script as data type which can then be interpreted natively or via a Plutus script. In the below example code, the type \( \text{MultiSigScript} \) is defined as a tree-structure \( \text{MultiSigTerm} \) that is either a single signature leaf node or a list of \( \text{MultiSigTerms} \) and a threshold.

```haskell
module MultiSigScript where

import qualified Data.Set as Set
import Data.Set (Set)
```
\[(\text{utxo, \_\_}) := \text{utxoSt}\]

\[
\forall \text{inp} \mapsto \text{validator} \in \text{validators} (\text{txinsScript} (\text{txins tx})) \text{ utxo} (\text{txwitsScript} \text{ tx}), \\
\text{validator} () () (\text{pendingTx} \text{ tx})
\]

\[
|\text{validators} (\text{txinsScript} (\text{txins tx})) \text{ utxo} (\text{txwitsScript} \text{ tx})| = |\text{txinsScript} (\text{txins tx}) \text{ utxo}|
\]

\[
\forall \text{vk} \mapsto \sigma \in \text{txwitsVKey tx}, \forall \text{x}{\text{txbody tx}}\sigma, \\
\text{witsNeeded utxo tx} \subseteq \{\text{hashKey vk | vk} \in \text{dom} (\text{txwitsVKey tx})\}
\]

\[
\text{utxoSt } \vdash \text{utxoEnv } \Rightarrow \text{utxoSt}'
\]

\[
\text{utxoSt } \vdash \text{utxoEnv } \Rightarrow \text{utxoSt}'
\]

**Figure 4:** UTxO with Witnesses and Multi-Sig

data VKeyHash = VKeyHash Int -- dummy representation
    deriving (Eq, Ord, Show)

type MultiSigScript = MultiSigTerm

data MultiSigTerm = SingleSig VKeyHash
    | MultiSig Int [MultiSigTerm]
    deriving Show

The validation of such a term can then be done as follows, checking for presence of a single
signature as the base case and comparing the number of validating nodes in the list with the
threshold in the recursion case.

evalMultiSigScript :: MultiSigScript -> Set VKeyHash -> Bool

evalMultiSigScript (SingleSig vkh) vkhs =
    vkh ‘Set.member’ vkhs

evalMultiSigScript (MultiSig m ts) vkhs =
    sum [ if evalMultiSigScript t vkhs then 1 else 0 | t <- ts ] >= m

verifyCases :: MultiSigScript -> [(Set VKeyHash, Bool)] -> Bool

takeCases script cases =
    and [ evalMultiSigScript script keyset == expected
                  | (keyset, expected) <- cases]

The Appendix A provides some examples of how this scheme can be used. The intention is
to have an alternative in the case for whenever the simple Plutus script integration would not
be viable.

The integration would follow the same approach as outlined for Plutus scripts in Sections 2
and 3. First, a new type of address is introduced which holds the hash of the necessary
information about how an output can be unlocked. Then, a new witness type is defined and
added to the transaction type definition. After that, the STS rule UTXOW is extended with the
validation of the new witnesses.
5 Summary

The presented multi-signature base on Plutus scheme does not require the scripts to use any cryptographic primitives. It requires only the ability to compare the required keys to those that actually signed the transaction. Gas cost can therefore be calculated statically in advance. The scripts could be realized as smart contracts with only limited requirements on functionality on the script language.

The necessary extensions to the data types in the Shelley specification Corduan et al. (2019) are relatively simple. They consist mainly of the introduction of additional optional data for scripts or hashed scripts.

The relaxation on accepting a superset of the strictly required signatures allows the creation of transactions with an arbitrary number of signatures. This potentially is a risk for an attack. The number of signatures should be taken into account in some way in the calculation of the transaction fee.

The alternative, script-like integration as DSL allows for native integration via the direct interpretation of the MultiSigScript. Its integration works in the same way as the Plutus script-based approach. If the Plutus approach is not viable for any reason, e.g., script size or readiness of library, it can be pursued instead.

References


A Native Multi-Signature Examples

mkVKeyHashSet :: [Int] -> Set VKeyHash
mkVKeyHashSet = Set.fromList . map VKeyHash

example1Of2 =
  MultiSig 1 [SingleSig (VKeyHash 1), SingleSig (VKeyHash 2)]

eexample1Of2_verify =
  verifyCases
    example1Of2
    [ (mkVKeyHashSet [], False)
      , (mkVKeyHashSet [1], True)
      , (mkVKeyHashSet [2], True)
      , (mkVKeyHashSet [1,2], True)
      , (mkVKeyHashSet [3], False)
    ]

eexample20f2 =
MultiSig 2 [SingleSig (VKeyHash 1), SingleSig (VKeyHash 2)]

example2Of2_verify =
verifyCases
  example2Of2
  [ (mkVKeyHashSet [], False),
    (mkVKeyHashSet [1], False),
    (mkVKeyHashSet [2], False),
    (mkVKeyHashSet [1,2], True),
    (mkVKeyHashSet [3], False),
    (mkVKeyHashSet [1,2,3], True)
  ]

exampleNestedOrAnd =
  MultiSig 1 [ MultiSig 2 [SingleSig (VKeyHash 1), SingleSig (VKeyHash 2)]
               , MultiSig 2 [SingleSig (VKeyHash 3), SingleSig (VKeyHash 4)]
  ]

exampleNestedAndOr =
  MultiSig 2 [ MultiSig 1 [SingleSig (VKeyHash 1), SingleSig (VKeyHash 2)]
               , MultiSig 1 [SingleSig (VKeyHash 3), SingleSig (VKeyHash 4)]
  ]

exampleNestedOrAnd_verify =
verifyCases
  exampleNestedOrAnd
  [ (keysset, expected)
    | has1 <- [False, True]
    , has2 <- [False, True]
    , has3 <- [False, True]
    , has4 <- [False, True]
    , has5 <- [False, True]
    , let keysset = mkVKeyHashSet $ [ 1 | has1 ]
      ++ [ 2 | has2 ]
      ++ [ 3 | has3 ]
      ++ [ 4 | has4 ]
      ++ [ 5 | has5 ]
    
    expected = (has1 && has2)
      || (has3 && has4)
  ]

exampleNestedAndOr_verify =
verifyCases
  exampleNestedAndOr
  [ (keysset, expected)
    | has1 <- [False, True]
    , has2 <- [False, True]
    , has3 <- [False, True]
    , has4 <- [False, True]
    , has5 <- [False, True]
    , let keysset = mkVKeyHashSet $ [ 1 | has1 ]

++ [ 2 | has2 ]
++ [ 3 | has3 ]
++ [ 4 | has4 ]
++ [ 5 | has5 ]

expected = (has1 || has2)
\&\& (has3 || has4)