Message-passing in the Extended UTxO Ledger

Polina Vinogradova[1][0000−0003−3271−3841] and Orestis Melkonian[2][0000−0003−2182−2698]

[1] Input Output, Canada (IOG), firstname.lastname@iohk.io
[2] Input Output, United Kingdom (IOG), firstname.lastname@iohk.io

Abstract. A notable problem faced by developers of smart contracts running on an extended UTxO (EUTxO) ledger is *double satisfaction*: interacting contracts that make payouts may validate with insufficient payments made to some recipients. In this work, we formalize the notion of a stateful contract constraint being vulnerable to double satisfaction. Next, we formalize interaction among scripts and stateful contracts via *message-passing*, consisting of a specification and an implementation of a stateful distributed message-passing contract, together with a proof of the integrity of its implementation. Messages specify sender and receiver outputs, as well as the data and assets being communicated, which are recorded on the ledger in the form of special NFT tokens distributed across UTxO entries that also contain the sent assets. We give two applications of our design by considering a message: (1) as a record of a successful script computation, akin to memoization, and (2) as a mechanism for asynchronous structured contracts communication that enables a principled separation of contract communication from its computation. Building on this application of message-passing, we present a result stating that making payouts from stateful contracts using message-passing is not vulnerable to double satisfaction.

Keywords: Blockchain · Ledger · UTxO · EUTxO · Smart contract · Formal verification · Small-step operational semantics · Message-passing · Double satisfaction · Simulation relation

1 Introduction

Message-passing is the standard for communicating data and assets between contracts in account-based ledgers [3,26,14]. An alternative to the account-based ledger model is the EUTxO (extended UTxO) ledger model, implemented by platforms such as Cardano [7,16] and Ergo [10]. It is a smart contract- (or *script*-) enabled UTxO-based ledger model, where user-defined script code is used to specify conditions a transaction must satisfy to be permitted to spend a UTxO entry or mint a token. Communication between scripts in EUTxO-based ledgers follows a different architecture than for account-based models. A script may require that another script executes successfully within the same transaction. Script interaction and communication is implemented using these kinds of *script dependencies*, as well as other constraints on the script-executing transaction.

Relying on unstructured, ad-hoc communication among EUTxO scripts presents some challenges, in particular, in terms of amenability to formal verification. For example, in order for a stateful contract to make progress, in addition to executing scripts implementing the contract, it may be necessary to run arbitrary collections of scripts on which the contract update depends (either directly or via a sequence of dependencies). Another challenge is tracking the flow of assets and data, including marking certain quantities of assets or data as "from" a particular contract (e.g. a payout to a specific address), or "to" a particular contract (e.g. a pay-in from an address). Asset flow tracking is a special case of the more general *double satisfaction problem* (DSP).

The DSP may occur when multiple scripts within the same transaction share a constraint satisfied by the transaction. Problematic occurrences of DS are due to the lack of a mechanism to associate the fulfillment of a constraint with the script imposing said constraint. For example, a payout is made only once, satisfying two scripts, each of which was expecting a separate payout. Because many contracts make payouts, this is a widely discussed problem in EUTxO programming. In Section 3, we present a formalization of the DSP, which has not previously been formalized.

Previous work presents principled approaches to building stateful contracts in the EUTxO ledger model, such as the constraint-emitting machine design pattern [6], as well as the more general structured contract framework (SCF) [22]. These have been mechanized in the Agda proof assistant [20] and provide the conceptual basis for the actual specification of the Cardano ledger specification [7,16], which is formulated with small-step semantics and is also mechanized in Agda [15].

We present an EUTxO layer-2 implementation of *asynchronous message-passing* as a principled approach to communication among individual scripts and the stateful contracts they implement. Our message-passing architecture is a stateful contract constructed as an instance of the structured contract framework.

The state of the message-passing contract is a set of messages. On the ledger, a special NFT token encodes a single message in the contract state, and individual message tokens are distributed across distinct UTxO entries. Each message specifies *sender* and *recipient* UTxO entries, which are authenticated at the time of minting (burning, resp.) of the message token. The message token specifies the data being sent, and must be placed in a UTxO also containing the assets being sent. Any script is able to interface with the message-passing contract so long as (i) the user input to the script can be decoded as a list of messages being produced and consumed, and (ii) the contract ensures the minting and burning of the message NFTs corresponding to the messages it sends or receives.

Our decentralized stateful contract design constitutes a way to interpret the notion of message-passing communication on an EUTxO ledger. The scripts implementing this design facilitate concurrent updates to asset token balances on the ledger that exist independently of a shared database. We argue that the message-passing approach to script and stateful contract communication addresses some of the challenges of writing scripts to run on an EUTxO ledger.

To that end, we present two use cases of message-passing together with formal specification and verification of properties related to the integrity of their behaviour. We demonstrate how expressing payouts as messages from a stateful contract can resolve the DSP for payouts. The main contributions of this work are :

(i) formalization of the double satisfaction problem (Section 3);
(ii) specification and implementation of a message-passing structured contract (Section 4);
(iii) an application of the message-passing contract for memoization, including proven properties of its behaviour (Section 5.1);
(iv) an application of the message-passing contract as a means of asynchronous communication of data and assets between structured contracts, which serves as an alternative to communication via ad-hoc dependencies. We formalize properties of this application, including resilience of payout messages to the DSP (Section 5.2).

We note that an extended version of this paper, containing additional results, proofs, pseudocode, and examples, is available at [3].

2 Background

2.1 The EUTxO ledger model

First, we give an overview of the semantics we use for our contract and ledger specifications, introduced in prior work [6,4,22], but included here for the sake of self-containment.

Ledger types. For the purposes of self-containment, we include a description of EUTxO ledger model types. We note and justify the (minimal) changes we introduce to the existing model in the description. The types of booleans, natural numbers, and integers are denoted by $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{Z}$, respectively. The type $\mathsf{Ix} := \mathbb{N}$ is used for indexing, e.g. of elements in a list. The type $\mathsf{Slot} := \mathbb{N}$ is used to indicate blockchain time.

The ledger state consists of a UTxO set, which is a collection of unspent outputs, each associated with a unique identifier. An output is a triple $(a, v, d) \in \mathsf{Output}$, where $a \in \mathsf{Script}$ is the address of the output, $v \in \mathsf{Value}$ is the collection of assets at this address, and $d \in \mathsf{Datum}$ is data specified by the user at the time of constructing the transaction which creates this output (we give more details on these three types below). The type of the UTxO set is $\mathsf{UTxO} := \mathsf{OutputRef} \mapsto \mathsf{Output}$, which is a finite key-value map with unique keys of type $\mathsf{OutputRef}$. We denote a single element in a finite key-value map u (such as the UTxO set) by $i \mapsto o \in u$. An output reference $(tx, ix) \in \mathsf{OutputRef} := \mathsf{Tx} \times \mathsf{Ix}$ pointing to an output o in a UTxO set consists of the transaction tx, which added $(tx, ix) \mapsto o$ to the UTxO set, and the index ix, which is the position of output o in the list of outputs of tx.

[3] https://fc24.ifca.ai/wtsc/WTSC24_2.pdf

The type Value is used to represent bundles of multiple kinds of assets Each type of asset in the bundle $v \in \mathsf{Value}$ has a unique asset ID, $a \in \mathsf{AssetID} := \mathsf{Policy} \times \mathsf{TokenName}$, which identifies a class of fungible tokens. Associated to the asset ID of each type of asset in a bundle is a a quantity $q \in \mathsf{Quantity} := \mathbb{Z}$, specifying the amount of the asset with the given ID in v. When v contains 0 of a given asset type, its asset ID is not included in v. An asset bundle containing one kind of asset with asset ID $(\mathsf{policy},\ \mathsf{tokenName})$ of quantity one is denoted by $\{\ \mathsf{policy} \mapsto \{\mathsf{tokenName} \mapsto 1\}\ \}$.

An asset with ID (p, t) has the minting (and burning) policy $p \in \mathsf{Policy} := \mathsf{Script}$. When an asset under this policy is minted or burned, the policy script is executed to determine whether the transaction is allowed to perform this action. The token name t is specified by the user at the time of constructing the minting transaction. It is used to differentiate between assets under the same policy. Unlike previous work [4,22], where the token name is a string, we take $\mathsf{TokenName} := \mathsf{Data}$. Value forms a group under addition $(+)$ with a zero element (0) and a partial order $(\leq)$ [5].

A script $s \in \mathsf{Script}$ is a piece of user-defined code that is executed as part of transaction validation, applied to specific inputs. Script code is stateless and produces a boolean output. Scripts are executed as part transaction validation to check that a transaction is permitted to do the action with which the script is associated. Scripts are used to specify permissions for two kinds of actions: spending an output (these are referred to as "validators", or sometimes the "address" of the output), and minting or burning tokens (these are called minting policies).

We denote script application by $[\![\_]\!]$, followed by the script arguments. At the time of evaluation, the arguments supplied to a script consist of transaction data (of the transaction executing it), as well as the data about the specific action for which the script specifies permission (i.e. the output being spent, or the tokens being minted under the policy). An extra piece of data $d \in \mathsf{Redeemer}$, associated with the particular action being validated, is specified by the user at the time transaction construction.

On-chain data of variable type, including Datum, Redeemer, and TokenName, are all type synonyms for Data; for the sake of brevity, we will omit explicit calls to the corresponding encoding/decoding functions as these will be obvious from the types involved, so any time a value is used as Data presupposes that decoding is successful.

Updates to the ledger state are specified in the form of a Tx (transaction) data structure. A transaction $tx \in \mathsf{Tx}$ contains (i) a set of *inputs*, each containing an output reference, an output, and the associated redeemer, (ii) a list of outputs, which get entered into the UTxO set with the appropriately generated output references, (iii) a pair of slot numbers representing the validity interval of the transaction, (iv) a Value being minted by the transaction, (v) a redeemer for each of the minting policies being executed, and (vi) the set of (public) keys that signed the transaction, alongside their signatures.

Small-step specifications. We formulate the transitions of ledgers and contracts in the form of small-step operational semantics [21], as exemplified by the official specification of the Cardano ledger [7,16]. In our specifications and contract implementation pseudocode, we follow standard set-theory notation, and clarify any non-standard notation usage alongside it.

A transition relation $\mathsf{TRANS} \subseteq (\mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$ is a collection of 4-tuples. A member $(env,\ s,\ i,\ s')$ of this relation is also denoted by :

$$env \vdash s \xrightarrow[\text{TRANS}]{i} s'$$

The variable $env \in \mathsf{Env}$ is the environment of the state transition, $s \in \mathsf{State}$ is the starting state, $i \in \mathsf{Input}$ is the input, and $s' \in \mathsf{State}$ is the end state. The system TRANS is a labelled transition system. For a given transition $(env,\ s,\ i,\ s') \in \mathsf{TRANS}$, the pair $(env,\ i)$ of an environment and an input make up the *label* of this transition from s to s'. Conventionally [7], env is block-level data, such as blockchain time, whereas i is specified by the user, e.g. a transaction.

Ledger transition semantics. The ledger semantics on top of which we build the results of this paper are found in existing work [6,4,22], but we include them here for self-containment and in order to introduce appropriate notation. The ledger transition system is given by the subset $\mathsf{LEDGER} \subseteq \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \times \mathsf{UTxO}$. Membership in this subset is specified by a single transition rule ApplyTx, which ensures that $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$ whenever $\mathsf{checkTx}\ (slot,\ utxo,\ tx) = \mathsf{True}$, and $utxo'$ is given by $(\{\ i \mapsto o \in utxo \ \mid\ i \notin tx.\mathsf{outputRefs}\ \}) \cup tx.\mathsf{outputs}$. Here, the notation $\mathsf{r}.f$ represents accessing a (named) field f of a record r. This is expressed in rule ApplyTx below, where any unbound variables are implicitly considered as universally quantified.

$$utxo' \ :=\ (\{\ i \mapsto o \in utxo \ \mid\ i \notin tx.\mathsf{outputRefs}\ \}) \cup tx.\mathsf{outputs}$$

$$\textsc{ApplyTx}\ \frac{\mathsf{checkTx}\ (slot,\ utxo,\ tx)}{slot \vdash (\ utxo\ ) \xrightarrow[\text{LEDGER}]{tx} (\ \boldsymbol{utxo'}\ )}$$

The function $\mathsf{checkTx} : \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \to \mathbb{B}$ checks the predicates which are consistent with the EUTxO model on which this work builds [4,22]. This includes executing all required validator and minting policy scripts with the appropriate inputs. The projection $tx.\mathsf{outputRefs}$ returns a UTxO set containing an entry $k \mapsto o$ for each input i of tx, where the key of the entry is the output reference k of i, and its value is the output o of i. The value $utxo'$ is calculated by removing the UTxO entries in $utxo$ corresponding to those in $tx.\mathsf{outputRefs}$, and adding the entries constructed by tx, see [22] for details.

2.2 Structured contracts

The structured contract framework [22] is a formalism for specifying and demonstrating the integrity of the implementation of a stateful contract on LEDGER.

We give the definition here for self-containment and in order to introduce the appropriate notation. A structured contract includes a small-steps semantics specification, as well as a ledger representation of its state and input. The ledger representation is a pair of functions: one which computes the contract state from a given UTxO state (or fails), and another which computes the input to the contract for a given transaction.

For a given valid LEDGER step, the representation functions must compute a valid step in the structured contract specification given that the starting UTxO state corresponds to a contract state. This integrity constraint is expressed as a proof obligation for the instantiation of a structured contract. This design guarantees that no invalid contract state updates are ever possible on the ledger.

Suppose $\mathsf{STRUC} \subseteq (\{\star\} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$ is a small-step transition system. Let $\pi : \mathsf{UTxO} \to \mathsf{State} \cup \{\star\}$ and $\pi_{\mathsf{Tx}} : \mathsf{Tx} \to \mathsf{Input}$ be functions such that :

$$\frac{\pi\ u\ \neq\ \star \qquad e \vdash (u) \xrightarrow[\text{LEDGER}]{t} (u')}{(\pi\ u'\ \neq\ \star)\ \wedge\ \star \vdash (\pi\ u) \xrightarrow[\text{STRUC}]{\pi_{\mathsf{Tx}}\ t} (\boldsymbol{\pi\ u'})}$$

The triple $(\mathsf{STRUC}, \pi, \pi_{\mathsf{Tx}})$ is called a *structured contract*, and we denote it by $(\mathsf{STRUC}, \pi, \pi_{\mathsf{Tx}}) \succeq \mathsf{LEDGER}$. Note that π is function with an output that is a *maybe* type, $\mathsf{State} \cup \{\star\}$, where $\{\star\}$ is a singleton. When $\pi\ u = \star$, there is no contract state corresponding to the ledger state u. The block-level data is never exposed to user-defined scripts in this model, so that the context of a structured contract is necessarily $\star \in \{\star\}$.

3 The problem of double satisfaction

In the EUTxO model, scripts place constraints on the transactions executing them. *Multiple scripts* may place the *same constraint* on the data of a given transaction. The issue with certain undesirable instances of this situation is called the *double satisfaction problem* (DSP). The DSP has been discussed in Plutus documentation,[4] and in the context of contract audits,[5][6] but has not yet been formally analyzed.

The DSP applies to scripts and structured contracts that require transactions to make payouts, so it is frequently encountered in EUTxO script programming. A *naive* payout is a constraint of the form "the transaction must include an output containing value v, with address a". When two structured contract implementations both place such a constraint on a transaction, it may be satisfied by a single transaction output $(a, v, \_)$, resulting in insufficient payment made to a. Note here that we use the notation _ to represent a term whose value is not relevant to the computation in which it appears.

[4] https://plutus.readthedocs.io/en/latest/reference/writing-scripts/common-weaknesses/double-satisfaction.html
[5] https://medium.com/@vacuumlabs_auditing/cardano-vulnerabilities-1-double-satisfaction-219f1bc9665e
[6] https://github.com/tweag/tweag-audit-reports/blob/main/Marlowe-2023-03.pdf

To formalize this kind of vulnerability, let us assume that all systems discussed in this section are deterministic (i.e. have exactly one end state for each pair of input and start state), and define the following function, which returns all pairs of states in all valid transitions of a given structured contract STRUC :

$$\mathsf{s}\ \mathsf{STRUC} = \{\ (s, s')\ \mid\ \exists\, i,\ (\star, s, i, s') \in \mathsf{STRUC}\ \}$$

Definition (transition constraint). A *constraint* of a transition system STRUC is a subset $\mathsf{C} \subseteq \{\star\} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State}$ such that $\mathsf{STRUC} \subseteq \mathsf{C}$.

Definition (double satisfaction). A structured contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$ is *vulnerable to double satisfaction* with respect to a constraint C whenever there exists another contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC}')$, with $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$ and $\mathsf{s}\ \mathsf{STRUC} = \mathsf{s}\ \mathsf{STRUC}'$, such that $\mathsf{STRUC}' \cap \mathsf{C} \subsetneq \mathsf{STRUC}'$.

Example (TOGGLE *with extra constraint).* Consider a $(\pi, \pi_{\mathsf{Tx}}, \mathsf{TOGGLE})$ contract, with $\mathsf{State} := \mathbb{B}$ (i.e. Boolean), and $\mathsf{Input} := (\mathsf{toggle} \cup \{\star\}) \times \mathsf{Interval}[\mathsf{Slot}]$.

$$\textsc{NoOp}\ \frac{}{\vdash (x) \xrightarrow[\textsc{TOGGLE}]{(\star, \_)} (x)} \qquad \textsc{Toggle}\ \frac{5 \le j < k \le 9}{\vdash (x) \xrightarrow[\textsc{Toggle}]{(toggle, [j,k])} (\neg\, x)}$$

We define a contract TOGGLE' by removing $5 \le j < k \le 9$ from rule $\textsc{Toggle}$, assume it has the same projections π, π_{Tx} as TOGGLE, and define the constraint

$$\mathsf{C}(\star, \_, (t, [j,k]), \_) := (t = \mathsf{toggle}) \Rightarrow (5 \le j < k \le 9)$$

The contracts TOGGLE' and TOGGLE transition between the same states: $\mathsf{s}\ \mathsf{TOGGLE} = \mathsf{s}\ \mathsf{TOGGLE}' = x \mapsto x,\ x \mapsto \neg\, x$. Note that $\mathsf{TOGGLE} = \mathsf{TOGGLE}' \cap \mathsf{C} \subsetneq \mathsf{TOGGLE}'$, hence TOGGLE is vulnerable to DS with respect to C.

Discussion. Vulnerability to the DSP comes from the lack of association of some property of transaction data with a *specific structured contract* (or script implementing it) that requires this property to hold. Our DS definition formalizes the association between a transaction property and a contract by defining a property to be associated with a contract only in the case when the a property can be expressed as a predicate on the contract state update. Thus, any property not expressible as a property of a contract state update is vulnerable to DS.

This defines a broad class of constraints that are vulnerable to DS, but for which this vulnerability may not necessarily be problematic. The preceeding example demonstrates an unproblematic constraint, C, which only requires that a toggle action happens at a particular time interval. It is possible that multiple contracts require that some other action be performed in that same time interval simultaneously.

The onus is on the structured contract author to determine for which constraints placed on the transaction by the contract double satisfaction vulnerability

is a problem. Then, changes must be made to the contract to ensure that the vulnerability is removed. A general approach to mitigating arbitrary instances of DSP vulnerability is outside the scope of this work, however, in Section 6, we propose a solution to the DSP for payouts. A similar approach can be taken for DSP vulnerability for pay-ins. It is possible to define classes of contracts that are never vulnerable to DS. See the extended version of this work for a proof of the following lemma:

Lemma (DS-free contracts). A deterministic structured contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$ is not vulnerable to double satisfaction with respect to any constraint whenever for any (s, i), there exists an s' such that $(\star, s, i, s') \in \mathsf{STRUC}$.

DSP mitigation. An existing heuristic for addressing the DSP is to include a constraint in the implementing script(s) that forces them to fail if *any other scripts* are being run by the transaction. This effectively mitigates negative consequences of potential vulnerabilities of the given contract's constraints to the DSP. This is likely not a practical solution in many cases, however, as it is too restrictive. We note that, like the above example, this constraint is not on the contract state update, but rather, on the transaction. This means that it is itself vulnerable to DS. However, vulnerability of this constraint to DS will likely not be deemed to be a problem by script authors, as the purpose of introducing it is to mitigate the negative consequences of other constraints' vulnerabilities.

4 Message-passing in EUTxO

Conceptually, a message is data sent from a sender to a recipient [1]. In our design, a message is a data structure of type Msg encoded on the ledger in a specific way. It also includes a sender, receiver, and some data or assets. The content of $m \in \mathsf{Msg}$ is encoded as the TokenName of an NFT with the minting policy msgsTT. It is encoded as such in order to maintain certain guarantees about the message's integrity, which are ensured by the NFT minting policy. A message $m \in \mathsf{Msg}$ consists of the following fields :

(i) an output reference inUTxO : OutputRef. An output with this reference must be spent when the message token is minted;
(ii) an index msgIx : Ix. It is used to uniquely identify a message whenever multiple messages are produced in association with spending a single UTxO entry;
(iii) an output msgTo : Output. It is an output that must be spent to validate the consumption of the message *by that recipient* (such an output may not be unique);
(iv) an output msgFrom : Output. It is an output that must be spent to validate production of the message *by that sender*. Specifically, the entry $m.\mathsf{inUTxO} \mapsto m.\mathsf{msgFrom}$ must be spent;
(v) a value msgValue : Value. It specifies the assets being sent. When a message is minted and placed in an output, this output must *also* contain these assets;

(vi) data msgData : Data. It is the data being sent via this message.

Each message requires a unique identifier to enable some of the applications we present later. Here, we use an approach based on the thread token mechanism to ensure NFT uniqueness [4]. This mechanism requires that the thread token's minting policy checks that a particular output reference is spent from the UTxO by the minting transaction, and exactly one token is minted under this policy. To uniquely identify a message NFT we use the output reference inUTxO, together with the message index msgIx. Duplication of unique identifiers is forbidden by the implementing scripts.

Sending a list of messages is done by submitting a transaction that (i) **mints** the NFTs encoding each of the messages, and (ii) for each message, spends the sender output with a redeemer containing the list of messages "from" that output. For an output to receive a list of messages, a transaction must spend the outputs containing the messages, and **burn** the message tokens. It must also spend the receiver output, and supply it with a redeemer containing the messages it is receiving.

The state of the message-passing contract is given by a set of messages, with $\mathsf{State}_{\mathsf{MSGS}} := \mathbb{P}\ \mathsf{Msg}$, which represents messages that have been sent, but not yet received. Here, $\mathbb{P}\ \mathsf{Msg}$ denotes the set of all subsets of Msg. The input is the whole transaction, with $\mathsf{Input}_{\mathsf{MSGS}} := \mathsf{Tx}$. The MSGS transition system specifies the rules for sending and receiving messages, see Fig. 1.

The function

$$\mathsf{msgTkn}\ msg := \{\ \mathsf{msgsTT} \mapsto \{\ msg \mapsto 1\ \}\ \}$$

encodes a message as a message token, recording the message data as its token name, and msgsTT as its minting policy. According to this policy, each message token minted by a transaction must be placed into a UTxO entry locked by a special validator, msgsVal, which only checks that any message token in that UTxO entry is burned. The message token minting policy msgsTT performs the same checks and assignments (1, 3, 5, 6, 7) that are in the MSGS specification in Fig. 1, with the notable exception of checking the non-duplication of existing messages, as required by (2). This cannot be checked explicitly by msgsTT because it cannot inspect the global set of existing messages under this policy, and must instead be proved as a consequence of the generation of the message's unique identifier. The type of the decoded redeemer for both msgsTT and msgsVal is $\{\star\}$, as they are not used in the implementation.

The notation $[a1; ...; ak] : [A]$ represents a list of type A, with concatenation denoted by $+\!\!+$. The predicate _#_ takes two lists, returning True if they are disjoint, and $[\mathsf{f}\ a \mid a \leftarrow as]$ denotes list comprehension. The contracts msgsTT and msgsVal implementing the MSGS specification are given in Fig. 4 and 3.

The projection function π_{Msg} returns, for a given $utxo$, all messages encoded in the message tokens that exist in the UTxO set. It returns $\star$ when one or more messages have been duplicated or outputs incorrectly generated in the $utxo$.

This is guaranteed by msgOutsOK, see Fig. 2 for the details.

$$\pi_{\mathsf{Msg}}\ utxo := \begin{cases} \{\ m\ \mid\ \_ \mapsto o \in utxo, \mathsf{msgTkn}\ m\ \subseteq o.\mathsf{value}\} & \text{if } \mathsf{msgOutsOK}\ utxo \\ \star & \text{otherwise} \end{cases}$$

We give a proof sketch of the simulation relation between LEDGER and MSGS in the extended version of this paper. Recall that this relation ensures the integrity of the implementation, i.e. that the implementation of MSGS via the msgsTT and msgsVal scripts only allows ledger updates that are mapped to *valid* MSGS transitions (by the π and π_{Tx} projections).

(1) construct a list of messages encoded in redeemers

$$sndMsgs := [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ (i.\mathsf{redeemer}),\ sr = \mathsf{send}\ ]$$
$$rcvMsgs := [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ (i.\mathsf{redeemer}),\ sr = \mathsf{receive}\ ]$$

(2) check that no new messages are duplicates

$$[\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow usedInputs\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid m \leftarrow msgs\ ]$$

(3) compute the set of message token-containing outputs being created

$$newOuts\ :=\ \{\ (o,\ msg)\ \mid\ o\ \in\ tx.\mathsf{outputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ o.\mathsf{value}\ \}$$

(4) check that all the messages are correctly constructed : correct sender output, sender has correct redeemer, output reference is spent, one message per output, output containing message token has correct validator and sufficient value

$$\forall\ (o,\ msg)\ \in\ newOuts,\ (msg,\ (msg.\mathsf{inUTxO}, msg.\mathsf{msgFrom}, \_))\ \in\ sndMsgs$$
$$\wedge\ \{\ t\ \subseteq\ o.\mathsf{value}\ \mid\ \mathsf{dom}\ t\ =\ \{\mathsf{msgsTT}\}\ \} = \mathsf{msgTkn}\ msg$$
$$\wedge\ o.\mathsf{validator}\ =\ \mathsf{msgsVal}\ \wedge\ o.\mathsf{value}\ \geq\ msg.\mathsf{msgValue}$$

(5) compute the set of message token-containing outputs being spent

$$usedInputs := \{\ (i,\ msg)\ \mid\ i\ \in\ tx.\mathsf{inputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ i.\mathsf{output.value}\ \}$$

(6) check that all messages are correctly consumed :
the receiver output is correct, input has correct redeemer, and message exists

$$\forall\ (i,\ msg)\ \in\ usedInputs,\ (msg,\ (\_, msg.\mathsf{msgTo}, \_))\ \in\ rcvMsgs\ \wedge\ msg \in msgs$$

(7) check minting and burning of message tokens :

$$\Sigma_{(\_,msg)\ \in\ newOuts}\ \mathsf{msgTkn}\ msg\ +\ \Sigma_{(\_,msg)\ \in\ usedInputs}\ (-1)\ *\ (\mathsf{msgTkn}\ msg)$$
$$=\ \{\ \mathsf{msgsTT} \mapsto tkns\ \in\ tx.\mathsf{mint}\ \}$$

PROCESS

$$\star \vdash (\ msgs\ ) \xrightarrow[\text{MSGS}]{tx} \left(\begin{array}{r} (msgs \setminus [\ m \mid (\_,\ m) \leftarrow usedInputs\ ]) \\ \cup [\ m \mid (\_,\ m) \leftarrow newOuts\ ] \end{array} \right)$$

Fig. 1: Specification of the MSGS transition

5 Message-passing use cases

In this section we discuss applications of the message-passing structured contract.

5.1 Memoization

There may be strict resource use constraints that apply to executing code on a blockchain. It may not be possible for a transaction to run the code of a large contract in its entirety. It may be desirable to divide such code into less memory- and CPU-intensive functions whose outputs are pre-computed for use by an aggregate function. A script may not trust values pre-computed off-chain, so a proof that a value was correctly computed on-chain is required. In this section we describe a technique for constructing such proofs using the MSGS contract. It is similar to a specific kind of caching called *memoization* [12], which is also how we refer to our approach.

Consider a function $\mathsf{myFunction} : \mathsf{MyInType} \to \mathsf{MyOutType}$ which performs some computation. We define a script checkMyFunction (Fig. 5a), which wraps the computation done by myFunction. This script mints a message token with data $(\mathit{fIn}, \mathit{fOut})$, such that myFunction $\mathit{fIn} = \mathit{fOut}$, and a script useMyFunction (Fig. 5b) that can consume a message with the redeemer $[(\mathsf{receive}, m)]$ when m is addressed to an output locked by useMyFunction, and is sent by an output locked by checkMyFunction. This message serves as a proof that myFunction $\mathit{fIn} = \mathit{fOut}$, so, useMyFunction can perform a computation checkStuff relying on the fact that myFunction $\mathit{fIn} = \mathit{fOut}$. Note that msgTo is not constrained by this contract, so that the generated message can be addressed to any recipient.

We give the result that formalizes the use of message-passing to prove that myFunction $\mathit{fIn} = \mathit{fOut}$.

Lemma (Verified input-output pairs). For any $(s,\ u,\ tx, u') \in \mathsf{LEDGER}$, with $\pi\ u \neq \star$ and $(i,\ (\mathsf{useMyFunction}, v, d),\ r) \in tx.\mathsf{inputs}$, such that

$$\begin{aligned} [(\mathsf{receive},\ m)] &= r \\ (\mathit{fIn},\ \mathit{fOut}) &= m.\mathsf{msgData} \\ m.\mathsf{msgFrom} &= (\mathsf{checkMyFunction}, \_, \_) \end{aligned}$$

necessarily myFunction $\mathit{fIn} = \mathit{fOut}$, and $m.\mathsf{msgTo} = (\mathsf{useMyFunction}, v, d)$. For a proof sketch, see the extended version of the paper. Note here that the memoization approach we presented can be viewed as a kind of *untrusted oracle*. The computation done to produce the memoized input-output pair cannot be falsified, so that no trust is required to make use of it.

5.2 Contracts using message-passing

Stateful contract interaction, or communication, in the EUTxO model is implemented via dependencies [4]. A *dependency* of a script c is a constraint requiring

that another script c' must be executed within the same transaction, possibly with specific arguments. Using the MSGS contract to implement communication between contracts reduces ad-hoc reliance on arbitrary script dependencies, and makes contract interaction more principled and amenable to formal verification.

We say that stateful contracts *use message-passing* when they require the production or consumption of messages to or from scripts implementing the contract. We formalize this notion in this section. Note that due to space constraints, we omit several interesting results about contracts using message-passing, as well as a detailed example of a contract that makes payouts via messages, all of which can be found in the extended version of this work.

Message-passing specification is closely integrated with ledger semantics, and inspects the scripts, redeemers, and datums of the input transaction. Because of this, a message-passing contract must also inspect these in order to correctly construct a message. So, a state projection function for a contract that uses message-passing includes the UTxO entry relevant to the contract state, in full. The contract input is the complete transaction.

Suppose that $\mathsf{F} : \mathsf{Output} \mapsto \mathbb{B}$ is a constraint on outputs, and $\mathsf{c} : \mathsf{UTxO} \to \mathbb{B}$ is a constraint on a valid UTxO state. The contract denoted by $(\pi_{\mathsf{Fc}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC})$ is a structured contract with

$$\begin{aligned}
\mathsf{State} &:= \{ i \mapsto o \ \in\ u \ \mid\ u \in \mathsf{UTxO},\ \mathsf{F}\ o \ \} \\
\pi_{\mathsf{Fc}}\ u\ &:= \begin{cases} \{ i \mapsto o \ \in u \ \mid\ \mathsf{F}\ o \} & \text{if } \mathsf{c}\ u \\ \star & \text{otherwise} \end{cases} \\
\pi_{\mathsf{Tx}} &:= \mathsf{id}
\end{aligned}$$

We can combine STRUC and MSGS to construct the structured contract $\mathsf{STRUC}_{\mathsf{MSGS}}$,

$$\begin{aligned}
\pi_{\mathsf{State}-M}\ u\ &:= \begin{cases} (\pi_{\mathsf{Fc}}\ u, \pi_{\mathsf{Msg}}\ u) & \text{if } \ \pi_{\mathsf{Fc}}\ u\ \neq\ \star\ \neq\ \pi_{\mathsf{Msg}}\ u \\ \star & \text{otherwise} \end{cases} \\
\pi_{\mathsf{Tx}-M}\ &:=\ \mathsf{id}_{\mathsf{Tx}} \\
\mathsf{STRUC}_{\mathsf{MSGS}} &:= \{\ (\star, (s, m), tx, (s', m'))\ \mid\ (\star, s, tx, s') \in \mathsf{STRUC}, (\star, m, tx, m') \in \mathsf{MSGS}\ \}
\end{aligned}$$

We call this contract *message-augmentation* of STRUC. We define the following functions that filter messages sent or received by STRUC :

$$\begin{aligned}
\mathsf{getFromSTRUCmsgs}\ msgs &:= \{\ m\ \mid\ m\ \in msgs,\ \mathsf{F}\ (m.\mathsf{msgFrom})\ \} \\
\mathsf{getToSTRUCmsgs}\ msgs &:= \{\ m\ \mid\ m\ \in msgs,\ \mathsf{F}\ (m.\mathsf{msgTo})\ \}
\end{aligned}$$

Definition (Uses message-passing). We say that STRUC *uses message-passing* whenever the set defined by

$$\mathsf{getMSGS}\ (\star, (s, m), tx, (s'm')) \ :=\ \mathsf{getFromSTRUCmsgs}\ (m' \setminus m) \cup \mathsf{getToSTRUCmsgs}\ (m \setminus m')$$

is non-empty for some $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$.

We define the set of *payouts* in the step $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$ by

$$\mathsf{getPayouts}\ (\star, (s, m), tx, (s'm')) := \{msg \in \mathsf{getFromSTRUCmsgs}\ (m' \setminus m) \mid msg.\mathsf{msgValue} > 0 \wedge \neg\ (\mathsf{F}\ (msg.\mathsf{msgTo})\ \}$$

Whenever this set is necessarily non-empty for some step in $\mathsf{STRUC}_{\mathsf{MSGS}}$, we say that it *makes payouts with messages*.

Discussion. A contract is said to use message-passing whenever there is a step in $\mathsf{STRUC}_{\mathsf{MSGS}}$ that requires the production or consumption of a non-empty set of messages to or from STRUC. Some computation performed by contracts implementing STRUC may be contingent on receiving a specific message. For example, accepting a payment message sent by another contract.

Contracts that use message-passing share common features that are both necessary and sufficient for a script c implementing the contract to be able to interface with the message-passing contract : (i) the script's redeemer must decode to a list of sent/received messages, and (ii) the script must ensure that the corresponding messages are included in the transaction's mint field.

For a given step $(\star, s, t, s') \in \mathsf{STRUC}$, we refer to the messages sent and received by outputs that make up s, i.e. those filtered by F, as a script's *communication*. Calculating s' for the given (s, t) is the STRUC contract's computation. STRUC may still include arbitrary dependencies on scripts implementing contracts other than MSGS. Specifying when a contract has no non-message dependencies is important for determining when it is guaranteed to be able to progress. This is, however, the subject of future work.

6 Messages as payouts

A *payout* is a message that is from STRUC, but not addressed to STRUC, and specifies a sent value greater than zero. The function that returns all the payouts for a given contract, getPayouts, is a function of the start and end MSGS states *only*. Consider a transition $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$. To guarantee that a message-payout *msg* is made whenever rule $R \subseteq \mathsf{STRUC}_{\mathsf{MSGS}}$ applies, R must ensure that (i) $msg \in m' \setminus m$, (ii) msgValue $msg > 0$, and (iii) *msg* must be *from* an output locked by some script c implementing STRUC. The script c implementing this constraint of STRUC should instead include the constraint msgTkn $msg \in tx$.mint. For a detailed example of making payouts via messages, see the extended version of this work. Making payouts in this way has an advantage over the naive approach to payouts.

MSGS **payouts and double satisfaction.** In Section 3 we present a naive approach to payouts. This approach is vulnerable to DS, since the constraint requiring a payout to be made is expressed as a function the input transaction, rather than the state. Naive payout outputs can be produced and consumed by any valid transaction at any time, independently of the state update of any

contract. Without a mechanism to *associate a payout with its sender*, is not possible to include naive payouts in a contract's state.

Intuitively, making payouts via messages provides such a mechanism by ensuring that the sender of the payout is recorded in the message token, and that the message token has a unique identifier. Formally, since making payouts via messages can be expressed as a predicate on a the start and end MSGS states, rather than on the input transaction, constraints on message payouts are not vulnerable to DS for a message-enhanced contract. We can express this as follows (see Appendix A for the proof):

*Lemma (*MSGS*-payouts and DS)* Suppose $(\pi_{\mathsf{Fc}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC})$ is a structured contract, and $\mathsf{STRUC}_{\mathsf{MSGS}}$ is its message-enhanced version. Let $\mathsf{C} \supset \mathsf{STRUC}_{\mathsf{MSGS}}$ be a constraint expressible in terms of some predicate C' on the set of payout messages,

$$\mathsf{C}\ (\star, (s, m), tx, (s'm')) := \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}}\ (\star, (s, m), tx, (s', m')))$$

Then, the contract $\mathsf{STRUC}_{\mathsf{MSGS}}$ is not vulnerable to DS with respect to C.

7 Discussion

7.1 Related work

Message-passing is the backbone of distributed computing [1,8]. The π-calculus process calculus has been developed to formalize message-passing between processes in distributed computing scenarios [17]. We conjecture that it may be possible to apply this formalism to message-passing between structured contracts.

The UTxO ledger model introduced by Bitcoin [18], as well as EUTxO ledger implementations [10], are themselves message-passing schemes, wherein a transaction is a message to a script. Our scheme reinterprets messages in a way that allows them to have a single verified sender output, and a receiver that is also an output. The contract MSGS can be viewed as a kind of linear sub-ledger within LEDGER, which can be used as a tool in specification and verification of properties of communicating contracts.

In account-based ledgers [3,14,26], (synchronous) message-passing is the default mode of communication between contracts. The Scilla programming language [25], with its emphasis on separating communication from computation for stateful contracts on the Zilliqa ledger [26], inspired this work. Even though Scilla was developed for the account-based ledger model, the communicating automata structure it uses to model contracts may be useful in describing message-passing structured contracts as well.

Existing work on rigs [9], which are cryptographic data structures that provide integrity-at-a-distance, presents an approach to maintaining data integrity across potentially multiple state-managing machines. Aspects of this approach are similar in spirit to the thread-token technique we use to uniquely identify

messages and ensure non-duplication of message tokens on the ledger; both are based on temporal and causal dependencies of operations on one another.

A version of asynchronous, but centralized, message-passing is implemented in the ERC-20 Ethereum contract for fungible tokens [11]. The ERC-20 design is primarily for asset transfers, whereas ours can be used to communicate authenticated data as well. Implementing message-passing via a centralized data-storage contract such as ERC-20 on an EUTxO ledger would significantly increase contention over UTxO entries between message-passing transactions, and therefore reduce concurrency.

Formalization of blockchain and ledger functionality forms a foundation for rigorous reasoning about smart contracts security, discussed in the detailed overview [24]. Mathematical models of EUTxO and UTxO ledgers and smart contracts on those ledgers, including ours, often specify a simplified version of actual implementations [6,4,13,19,2,23].

7.2 Future work

The scheme we presented in Fig. 1 is such that the outputs that must be spent in order to consume a given message are fully specified (via the msgTo field of the message), including their scripts, values, and datums. In future work, this constraint could be relaxed for a more permissive and versatile system design. A *time of expiry* can be added to the message structure and used to specify a time after which a message can be consumed under different constraints. Changing the type of the message-passing redeemer from a list of messages to a list of messages (for communication) together with some extra data (for computation) can can allow a given script to more easily engage in both computation and communication as a result of applying a transaction.

In this work, we did not specify trace-based properties of LEDGER or any structured contract STRUC. This topic, in general, is the subject of future work. Of particular interest are structured contracts that can be guaranteed to take a step without the need for executing "external" scripts, i.e. ones other than those used to implement that contract. It may be unrealistic for a contract to always take a step without *any* external contracts validating, e.g. running a script which locks funds used for paying into the contract. However, it seems feasible to limit a structured contract's dependencies to message-passing only. Formalizing and proving properties about this class of structured contracts in the future is of interest.

In the future, we intend to mechanize this contract and its applications in Agda, building upon the formal EUTxO ledger model [6] and structured contracts framework [22]. To achieve the goal of integrating our work into the more realistic mechanization of the Cardano ledger [15], and eventually implement it on the platform, some adjustments to our design may be required.

7.3 Conclusion

Principled approaches to implementing and reasoning about the behavior of stateful smart contracts in the EUTxO ledger model have already been formalized in existing work. However, such models do not include any special provisions for analyzing communication among contracts. In this work, we focus on formalizing communication of data and assets among scripts as well as the stateful contracts they implement. We first formalize a common problem in contract interaction — the double satisfaction problem, which has to do with a single transaction satisfying the constraints of multiple scripts it executes. We demonstrate how a single pay-out made by a transaction in the case where a payment *per executed script* was expected may constitute an instance of the DS formalism we presented.

We define what a message data structure is in the context of an EUTxO ledger — a unique identifier associated with the data and assets being sent, as well as its sender and receiver outputs. We then define how to use the ledger asset-minting mechanism to encode messages as tokens which appear in the UTxO set. Messages are sent and received asynchronously, tracked by a distributed structured contract MSGS. A proof of correctness of its implementation guarantees that minted message tokens specify verified sender, receiver, data, and come with appropriate amounts of sent assets.

To give examples of formal reasoning about the message-passing contract and its applications, we present two use cases. The first is a variation on memoization, wherein message tokens specifying input-output pairs of a particular computation serve as proof artefacts of its correctness. The second formalizes the idea of structured contract communication via message-passing. We formalize when a message constitutes a "payout" from a contract, and then demonstrate how expressing payouts as messages can address vulnerability to the DSP in the case of payouts. The necessity of executing the MSGS-implementing scripts (which may require fee payment) whenever messages are sent and received is a limitation of our design.

Acknowledgments. We would like to thank Manuel Chakravarty for providing inspiration for this work, by being one of the first proponents of the message-passing idiom for concurrency in the EUTxO model. We would also like to thank Philip Wadler for useful discussions.

References

1. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)
2. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of Algorand smart contracts (2021)
3. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ (2014)

4. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P.: Native custom tokens in the extended UTXO model. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12478, pp. 89–111. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_7, https://doi.org/10.1007/978-3-030-61467-6_7
5. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P., Zahnentferner, J.: UTXO$_{ma}$: UTXO with multi-asset support. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12478, pp. 112–130. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_8, https://doi.org/10.1007/978-3-030-61467-6_8
6. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The extended UTXO model. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12063, pp. 525–539. Springer (2020). https://doi.org/10.1007/978-3-030-54455-3_37, https://doi.org/10.1007/978-3-030-54455-3_37
7. Corduan, J., Güdemann, M., Vinogradova, P.: A formal specification of the Cardano ledger. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf (2019)
8. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design (International Computer Science). Addison-Wesley Longman, Amsterdam (2005)
9. Coward, K., Toliver, D.R.: Simple rigs hold fast (2022)
10. Ergo Team: Ergo: A Resilient Platform For ContractualMoney. https://whitepaper.io/document/753/ergo-1-whitepaper (2019)
11. Ethereum Team: ERC-20 TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-20 (2023)
12. Field, A., Harrison, P.: Functional Programming. International computer science series, Addison-Wesley (1988), https://books.google.ca/books?id=nYtQAAAAMAAJ
13. Gabbay, M.J.: Algebras of UTxO blockchains. Mathematical Structures in Computer Science **31**(9), 1034–1089 (2021). https://doi.org/10.1017/S0960129521000438
14. Goodman, L.: Tezos—a self-amending crypto-ledger (white paper). https://tezos.com/whitepaper.pdf (2014)
15. Knispel, A., Melkonian, O., Chapman, J., Hill, A., Jääger, J., DeMeo, W., Norell, U.: Formal specification of the Cardano blockchain ledger, mechanized in Agda. https://omelkonian.github.io/data/publications/cardano-ledger.pdf (2024), under submission
16. Knispel, A., Vinogradova, P.: A Formal Specification of the Cardano Ledger integrating Plutus Core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf (2021)
17. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge, UK (1999)
18. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)

19. Nester, C.: A foundation for ledger structures. In: Anceaume, E., Bisière, C., Bouvard, M., Bramas, Q., Casamatta, C. (eds.) 2nd International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020, October 26-27, 2020, Toulouse, France. OASIcs, vol. 82, pp. 7:1–7:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.TOKENOMICS.2020.7, https://doi.org/10.4230/OASIcs.Tokenomics.2020.7
20. Norell, U.: Dependently typed programming in Agda. In: International School on Advanced Functional Programming. pp. 230–266. Springer (2008)
21. Plotkin, G.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61**, 17–139 (07 2004). https://doi.org/10.1016/j.jlap.2004.05.001
22. Polina Vinogradova and Orestis Melkonian and Philip Wadler and Manuel Chakravarty and Jacco Krijnen and Michael Peyton Jones and James Chapman and Tudor Ferariu : Structured contracts in the EUTxO ledger model (2024), https://fmbc.gitlab.io/2024/files/FMBC2024.pdf
23. Rupić, K., Rožić, L., Derek, A.: Mechanized Formal Model of Bitcoin's Blockchain Validation Procedures. In: Bernardo, B., Marmsoler, D. (eds.) 2nd Workshop on Formal Methods for Blockchains (FMBC 2020). Open Access Series in Informatics (OASIcs), vol. 84, pp. 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.FMBC.2020.7, https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2020.7
24. Sánchez, C., Schneider, G., Leucker, M.: Reliable smart contracts: State-of-the-art, applications, challenges and future directions. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. pp. 275–279. Springer International Publishing, Cham (2018)
25. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 185 (2019)
26. Team, T.Z.: The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf (2017)

A Proofs

*Proof of Lemma (*MSGS*-payouts and DS).* Suppose that $(\pi_{\mathsf{F,C}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC}')$ is another (more permissive) structured contract, with $\mathsf{STRUC}_{\mathsf{MSGS}} \subseteq \mathsf{STRUC}'_{\mathsf{MSGS}}$, and $\mathsf{s}\ \mathsf{STRUC}_{\mathsf{MSGS}} = \mathsf{s}\ \mathsf{STRUC}'_{\mathsf{MSGS}}$. For any $(\star,(s,m),tx,(s',m')) \in \mathsf{STRUC}'_{\mathsf{MSGS}}$, by definition,

$$\mathsf{getPayouts}_{\mathsf{STRUC}'}\ (\star,(s,m),tx,(s',m')) = \{\ ms \in m' \setminus m \mid \mathsf{F}\ (ms.\mathsf{msgFrom}) \wedge ms.\mathsf{msgValue} > 0 \wedge \neg\ (\mathsf{F}\ (ms.\mathsf{msgTo}))\ \}$$

which depends only on F (which is the same for STRUC and STRUC′), and $m' \setminus m$. Now, by the assumed preconditions on STRUC′, for any $(\star,(s,m),tx,(s',m')) \in \mathsf{STRUC}'_{\mathsf{MSGS}}$, we can find $(\star,(s,m),tx',(s',m')) \in \mathsf{STRUC}_{\mathsf{MSGS}} \subseteq \mathsf{STRUC}'_{\mathsf{MSGS}}$. Then,

$$\begin{aligned}\mathsf{C}\ (\star,(s,m),tx,(s'm')) &= \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}'}\ (\star,(s,m),tx,(s',m'))) \\ &= \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}}\ (\star,(s,m),tx',(s',m'))) \\ &= \mathsf{C}\ (\star,(s,m),tx',(s'm'))\end{aligned}$$

Therefore, any transition in $\mathsf{STRUC}'_{\mathsf{MSGS}}$ must also satisfy C. We get that $\mathsf{STRUC}'_{\mathsf{MSGS}} \cap \mathsf{C} = \mathsf{STRUC}'_{\mathsf{MSGS}}$, meaning that $\mathsf{STRUC}_{\mathsf{MSGS}}$ is not vulnerable to DS with respect to such a C.

B Pseudocode

$$
\begin{aligned}
&\mathsf{msgOutsOK} : \mathsf{UTxO} \to \mathbb{B} \\
&\mathsf{msgOutsOK}\ utxo := \\
&\qquad \forall\, (i \mapsto o) \in utxo, \{\ \mathsf{msgsTT} \mapsto \{m \mapsto q\}\} \subseteq o.\mathsf{value} \Rightarrow \\
&\qquad\quad (q = 1) \\
&\qquad\quad \wedge\ \ (m \neq \star)\ \wedge\ (m.\mathsf{inUTxO} \mapsto \_ \notin utxo) \\
&\qquad\quad \wedge\ \ [\![\mathsf{msgsTT}]\!]\ (\star,\ (i.\mathsf{id},\ \mathsf{msgsTT})) \\
&\qquad\quad \wedge\ \ \forall (i' \mapsto o') \in utxo, i \neq i',\ \{\ \mathsf{msgsTT} \mapsto \{m \mapsto \_\}\} \notin o'.\mathsf{value} \\
&\qquad \wedge\ \forall\, (tx, ix) \mapsto o\ \in\ utxo,\ \forall\, i \in\ tx.\mathsf{inputs}, \\
&\qquad\qquad [\![i.\mathsf{output.validator}]\!]\ (i.\mathsf{output.datum},\ i.\mathsf{redeemer},\ (tx,\ i)) \\
&\qquad\qquad \wedge\ \ (ix \mapsto o) \in\ tx.\mathsf{outputs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{SR} := \{\mathsf{send}, \mathsf{receive}\} \\
&\quad \textit{Tag specifying whether message is being sent or received}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{getMsgRef} : \mathsf{Msg} \to (\mathsf{OutputRef},\ \mathsf{Ix}) \\
&\mathsf{getMsgRef}\ msg := (msg.\mathsf{inUTxO},\ msg.\mathsf{msgIx}) \\
&\quad \textit{Returns unique message identifier}
\end{aligned}
$$

Fig. 2: Projections and auxiliary MSGS functions

$$
\begin{aligned}
&\mathsf{msgsTT} := \mathsf{msgsTT}'\ \mathsf{msgsVal} \\
&\\
&[\![\mathsf{msgsVal}]\!]\ (\_,\ \_,\ (tx,\ i))\ := \\
&\quad \forall\, msg \in\ \{\ \{\ m\ \mid\ \mathsf{msgsTT}'\ (i.\mathsf{output.validator})\ \mapsto\ \{\ m \mapsto 1\}\} \subseteq\ i.\mathsf{output.value}\ \}, \\
&\quad \{\ \mathsf{msgsTT}'\ (i.\mathsf{output.validator})\ \mapsto\ \{msg \mapsto -1\}\}\ \subseteq\ tx.\mathsf{mint}
\end{aligned}
$$

Fig. 3: Minting policy and validator for UTxO containing message tokens

$$
\begin{array}{l}
\mathsf{msgsTT'} : \mathsf{Script} \to \mathsf{Script} \\
[\![\mathsf{msgsTT'}\ mv]\!]\ (\_,\ (tx,\ pid)) \ :=\ \forall\ msg,\ \{\ pid\ \mapsto\ \{msg \mapsto \_\}\ \} \subseteq tx.\mathsf{mint}, \\
\quad [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow usedInputs\ ] \\
\quad \wedge\ \forall\ (o,\ msg)\ \in\ newOuts, \\
\qquad\quad (msg,\ (msg.\mathsf{inUTxO}, msg.\mathsf{msgFrom}, \_))\ \in\ sndMsgs \\
\qquad \wedge\ \{\ t\ \subseteq\ o.\mathsf{value}\ \mid\ \mathrm{dom}\ t\ =\ \{pid\}\ \} = \mathsf{msgTkn}\ msg \\
\qquad \wedge\ o.\mathsf{validator}\ =\ mv\ \wedge\ o.\mathsf{value}\ \geq\ msg.\mathsf{msgValue} \\
\quad \wedge\ \forall\ (i,\ msg)\ \in\ usedInputs,\ (msg,\ (\_, msg.\mathsf{msgTo}, \_))\ \in\ rcvMsgs \\
\quad \wedge\ \Sigma_{(\_,msg)\ \in\ newOuts}\ \mathsf{msgTkn}\ msg\ +\ \Sigma_{(\_,msg)\ \in\ usedInputs}\ (-1)\ *\ (\mathsf{msgTkn}\ msg)\ = \\
\qquad \{pid \mapsto tkns\ \in\ tx.\mathsf{mint}\ \} \\
\quad \textbf{where} \\
\qquad \mathsf{msgTkn}\ msg\ :=\ \{\ pid\ \mapsto\ \{msg \mapsto 1\}\ \} \\
\qquad sndMsgs\ \ := [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ i.\mathsf{redeemer},\ sr = \mathsf{send}\ ] \\
\qquad rcvMsgs\ \ := [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ i.\mathsf{redeemer},\ sr = \mathsf{receive}\ ] \\
\qquad newOuts\ \ := \{\ (o,\ msg)\ \mid\ o\ \in\ tx.\mathsf{outputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ o.\mathsf{value}\ \} \\
\qquad usedInputs := \{\ (i,\ msg)\ \mid\ i\ \in\ tx.\mathsf{inputs}\ tx,\ \mathsf{msgTkn}\ msg\ \subseteq\ i.\mathsf{output.value}\ \}
\end{array}
$$

Fig. 4: Minting policy constructor for message tokens

$$
\begin{array}{l}
[\![\mathsf{checkMyFunction}]\!]\ (\_,\ r,\ (tx,\ i)) \ := \\
\quad m.\mathsf{inUTxO}\ =\ i.\mathsf{outputRef} \\
\wedge\ m.\mathsf{msgFrom}\ =\ i.\mathsf{output} \\
\wedge\ m.\mathsf{msgValue}\ =\ 0 \\
\wedge\ \mathsf{msgTkn}\ m\ \subseteq\ tx.\mathsf{mint} \\
\wedge\ \mathsf{myFunction}\ fIn\ =\ fOut \\
\textbf{where} \\
[(\mathsf{send},\ m)]\ =\ r \\
(fIn,\ fOut)\ \ =\ m.\mathsf{msgData}
\end{array}
$$

(a) Script minting message token.

$$
\begin{array}{l}
[\![\mathsf{useMyFunction}]\!]\ (d,\ r,\ (tx,\ i)) \ := \\
\quad (\ \ m.\mathsf{msgFrom}\ =\ (\mathsf{checkMyFunction}, \_, \_) \\
\quad \wedge\ m.\mathsf{msgTo}\ =\ i.\mathsf{output} \\
\quad \wedge\ (-1)\ *\ (\mathsf{msgTkn}\ m)\ \subseteq\ tx.\mathsf{mint} \\
\quad \wedge\ \mathsf{checkStuff}\ d\ r\ \ (tx,\ i)\ (fIn,\ fOut)\ \ ) \\
\vee\ \mathsf{checkOtherStuff}\ d\ r\ \ (tx,\ i) \\
\textbf{where} \\
[(\mathsf{receive},\ m)]\ =\ r \\
(fIn,\ fOut)\ \ \ \ =\ m.\mathsf{msgData}
\end{array}
$$

(b) Script using the memoized output.

Fig. 5: Scripts for memoizing the output of myFunction.