Abstract—We propose a secure and efficient implementation of fungible tokens on Bitcoin. Our technique is based on a small extension of the Bitcoin script language, which allows the spending conditions in a transaction to depend on the neighbour transactions. We show that our implementation is computationally sound: that is, adversaries can make tokens diverge from their ideal functionality only with negligible probability.

Index Terms—Bitcoin, tokens, neighbourhood covenants

I. INTRODUCTION

One of the main applications of blockchain technologies is the exchange of custom crypto-assets, called tokens. Token transfers currently involve $\sim 50\%$ of the transactions on the Ethereum blockchain [1], and they are at the basis of many protocols built on top of that platform [2], [3]. Broadly, tokens are classified as fungible or non-fungible. Fungible tokens can be split into smaller units: different units of the same token can be used interchangeably. Further, users can join units of the same fungible token, and exchange them with other crypto-assets. Instead, non-fungible tokens cannot be split or joined.

Historically, the first implementations of tokens were developed before Ethereum, on top of Bitcoin. Some of them (e.g., [4]) used small bitcoin fractions to represent the token value; some others (e.g., [5]–[7]) embedded the token value in other transaction fields [8], to cope with the fluctuating bitcoin price. All these implementations have a common drawback: the correctness of the token actions is not guaranteed by the consensus protocol of the blockchain. In fact, the blockchain is used just to notarize the actions that manipulate tokens, but not to check that these actions are actually permitted. Typically, the owners of these tokens must resort to off-chain mechanisms (e.g., trusted authorities) to have some guarantees on the correct use of tokens, e.g. that they are not double-spent, or that distinct tokens are not joined.

By contrast, modern blockchain platforms support on-chain tokens, whose correctness is guaranteed by the consensus protocol of the blockchain. Some platforms (e.g., Algorand [9]) natively support tokens, while some others (e.g., Ethereum) encode them as smart contracts. Bitcoin, instead, does not support tokens natively, and its limited script language is not expressive enough to implement them as smart contracts. Since adding native tokens to Bitcoin appears to be out of reach, given the resilience of the Bitcoin community to radical changes [10], the only viable alternative is to devise a small, efficient extension of the script language which increases the expressiveness of Bitcoin enough to support tokens.

A recent Bitcoin Improvement Proposal (BIP 119 [11], [12]) aims at extending the Bitcoin script language with covenants, a class of operators that allow a transaction to constrain how its funds can be used by the redeeming transactions. Although these covenants enable a variety of use cases, e.g. vaults, batched payments, and non-interactive payment channels [11], they are not expressive enough to implement fungible tokens in a practical way. Roughly, the covenants proposed in the literature can check that the token units are preserved upon split actions, but they cannot ensure this property upon join actions (we describe these and other issues in Section II).

In this work, we propose a variant of covenants, named neighbourhood covenants, which can inspect not only the redeeming transaction, but also the siblings and the parent of the spent one. This extension preserves the basic UTXO design of Bitcoin, adding only a few opcodes to its script language, which is kept efficient, loop-free, and non Turing-complete. Still, neighbourhood covenants significantly increase the expressiveness of Bitcoin as a smart contracts platform, allowing to execute arbitrary smart contracts by appending a chain of transactions to the blockchain. Technically, we prove that neighbourhood covenants make Bitcoin Turing-complete.

Although this expressiveness result is of theoretical interest, in itself it does not enable an efficient implementation of tokens. To recover efficiency, we implement token actions in a single, succinct script which exploits neighbourhood covenants. We devote a large portion of the paper to establish the security of our construction: in brief, we define a symbolic model of token actions, and a computational model, where performing these actions corresponds to appending transactions to the Bitcoin blockchain. Our main technical result is a computational soundness theorem, which ensures that any execution in the computational model has a corresponding execution in the symbolic one. Therefore, we guarantees that a computational adversary cannot make the behaviour of tokens diverge from the behaviour of the symbolic model.

Contributions: We summarise our contributions as follows:

- we introduce a symbolic model of fungible tokens, which formalises their archetypal features: their minting and burning, the split and join operations, and the exchange of tokens with other tokens or with bitcoins (Section III);
- we propose neighbourhood covenants as a Bitcoin extension (Section V), and we show that they make Bitcoin Turing-powerful (Theorem 2). We then discuss how to efficiently implement them on Bitcoin;
• we exploit neighbourhood covenants to implement tokens on Bitcoin (Section VI);
• we introduce a computational model for Bitcoin and we prove the computational soundness of our token implementation (Theorem 5 in Section VII);
• as an consequence, we show that a value preservation property established in the symbolic model can be lifted for free to the computational model (Theorem 6).

The proofs of our results are in the Appendix.

II. OVERVIEW OF THE APPROACH

In this section we summarize our approach: in particular, we sketch our implementation of Bitcoin tokens, motivating the use of neighbourhood covenants to guarantee their security.

Tokens: We propose a symbolic model of fungible tokens. Since non-fungible tokens are the special case of fungible ones where each token is generated exactly in one unit, hereafter we consider the general case of fungible tokens. The basic element of our model is the deposit, i.e. a term of the form:

\[ \langle A, v : \tau \rangle_x \]

which represents the fact that a user \( A \) owns \( v \) units of a token \( \tau \), where \( \tau \) may denote either user-defined tokens or bitcoins (\( \mathbb{B} \)). The index \( x \) uniquely identifies the term within a configuration, i.e. a composition of deposits, e.g.:

\[ \langle A, 1 : \tau \rangle_x | \langle A, 2 : \tau \rangle_y | \langle B, 3 : \mathbb{B} \rangle_z \]

We define a few actions to mint and manipulate tokens. First, any user \( A \) can mint \( v \) units of a new token, spending a deposit of 0 \( \mathbb{B} \). Performing this action (say, with \( v = 10 \)) is modelled as a state transition, whose labels records the performed action:

\[ \langle A, 0 : \mathbb{B} \rangle_{x_0} \xrightarrow{\text{gen}} \langle A, 10 : \tau \rangle_{x_1} \]

(1)

where the identifier \( x_1 \) of the new deposit and the identifier \( \tau \) of the minted token are fresh. After performing the action, \( A \) owns a deposit of ten units of the token \( \tau \). As said before, one of the peculiar properties of fungible tokens is that they can be split. When splitting her deposit in two smaller deposits, \( A \) can choose the owner of one of the new deposits, e.g.:

\[ \langle A, 10 : \tau \rangle_{x_1} \xrightarrow{\text{split}} \langle A, 8 : \tau \rangle_{x_2} | \langle B, 2 : \tau \rangle_{x_3} \]

(2)

A user can transfer the ownership of any of her deposits to another user. For instance, \( A \) can give her deposit \( x_2 \) to \( B \):

\[ \langle A, 8 : \tau \rangle_{x_2} \xrightarrow{\text{give}} \langle B, 8 : \tau \rangle_{x_4} \]

(3)

After that, \( B \) owns a total of 10 units of \( \tau \) in two separate deposits, one with 8 units, and the other one with 2 units. This reflects the UTXO nature of Bitcoin: by contrast, in account-based blockchains like Ethereum, \( B \) would have a single account storing 10 units of \( \tau \). Now, \( B \) can join his two deposits, obtaining a single deposit with 10 units of \( \tau \). When performing the join action, \( B \) can also choose the owner of the new deposit, in this case transferring it back to \( A \):

\[ \langle B, 8 : \tau \rangle_{x_4} | \langle B, 2 : \tau \rangle_{x_3} \xrightarrow{\text{join}} \langle A, 10 : \tau \rangle_{x_5} \]

(4)

A crucial property of the join operation is that only deposits of the same token can be joined together. Thus, two deposits of \( \tau \) and \( \tau' \) with \( \tau \neq \tau' \) cannot be joined:

\[ \langle B, 8 : \tau \rangle_{x_4} | \langle A, 2 : \tau' \rangle_{x_6} \xrightarrow{\text{join}} \]

In this configuration, if both \( A \) and \( B \) agree, they can exchange the ownership of their tokens:

\[ \langle B, 8 : \tau \rangle_{x_4} | \langle A, 2 : \tau' \rangle_{x_6} \xrightarrow{\text{xchg}} \langle A, 8 : \tau \rangle_{x_7} | \langle B, 2 : \tau' \rangle_{x_8} \]

The xchg operation also supports the exchange between bitcoins and other tokens, representing the trade of tokens. For instance, \( A \) can buy 2 units of \( \tau' \) from \( B \) for 1\( \mathbb{B} \):

\[ \langle B, 2 : \tau' \rangle_{x_9} | \langle A, 1 : \mathbb{B} \rangle_{x_9} \xrightarrow{\text{xchg}} \langle A, 2 : \tau' \rangle_{x_{10}} | \langle B, 1 : \mathbb{B} \rangle_{x_{11}} \]

Bitcoin: Although Bitcoin does not support user-defined tokens, it implements all the operations discussed above on its native crypto-currency. Intuitively, each deposit corresponds to a transaction output, and performing actions corresponds to appending a suitable transaction that redeems it.

For instance, minting bitcoins is obtained through coinbase transactions, which are used in Bitcoin to pay rewards to miners. We represent a coinbase transaction as follows:

\[ \text{in}(1): \bot \]
\[ \text{wit}(1): \bot \]
\[ \text{out}(1): \{ \text{scr : versig}(pk_A, rtx.wit), \text{val : 10}\mathbb{B} \} \]

In general, the \text{in} field points to a previous transaction on the blockchain, that the current one is trying to spend. Here, the "undefined" value \( \bot \) characterizes \( T_1 \) as a coinbase, since it mints bitcoins without spending any transaction. The \text{out} field is a record, where \text{scr} is a script, and \text{val} is the amount of bitcoins that will be redeemed by a subsequent transaction which points to \( T_1 \) and satisfies its script. Here, the script \text{versig}(pk_A, rtx.wit) verifies a signature on the redeeming transaction (rtx, excluding its \text{wit} field) against \( A \)'s public key \( pk_A \). This signature is retrieved from the \text{wit} field of \text{rtx}. Since \( A \) is the only user who can redeem \( T_1 \), we can say that \( T_1 \) is the computational counterpart of the deposit \( \langle A, 10 : \mathbb{B} \rangle_{x_1} \).

To perform the \text{split} action (2) on \( \tau = \mathbb{B} \), we can spend \( T_1 \) with a transaction \( T_2 \) with two outputs:

\[ \text{in}(1): \{ \text{T}_1, 1 \} \]
\[ \text{wit}(1): \text{sig}_{sk_A}(T_1) \]
\[ \text{out}(1): \{ \text{scr : versig}(pk_A, rtx.wit), \text{val : 8}\mathbb{B} \} \]
\[ \text{out}(2): \{ \text{scr : versig}(pk_A, rtx.wit), \text{val : 2}\mathbb{B} \} \]

The first output, that we denote by \( (T_2, 1) \), corresponds to the deposit \( \langle A, 8 : \mathbb{B} \rangle_{x_2} \) in (2). Instead, the output \( (T_2, 2) \) corresponds to \( \langle B, 2 : \mathbb{B} \rangle_{x_3} \). These outputs can be spent independently. For instance, performing the \text{give} action in (3) corresponds to appending a transaction which spends \( (T_2, 1) \):

\[ \text{in}(1): \{ \text{T}_2, 1 \} \]
\[ \text{wit}(1): \text{sig}_{sk_A}(T_3) \]
\[ \text{out}(1): \{ \text{scr : versig}(pk_A, rtx.wit), \text{val : 8}\mathbb{B} \} \]
At this point, we have two unspent outputs on the blockchain: \((T_2, 2)\) and \((T_3, 1)\). We can perform the join action in (4) by spending both of them simultaneously with the following transaction, which has two inputs:

\[
\begin{align*}
\text{in}(1): & (T_2, 2) \\
\text{in}(2): & (T_3, 1) \\
\text{wit}(1): & \text{sig}_{sk_8}(T_4) \\
\text{wit}(2): & \text{sig}_{sk_9}(T_4) \\
\text{out}: & \{ \text{scr} \triangleright \text{versig}(pk_B, \text{rtx.wit}), \text{val} \triangleright 10\} \\
\end{align*}
\]

Implementing Bitcoin tokens with covenants: Although the Bitcoin script language is a bit more flexible than shown above, it does not allow to implement on-chain tokens. One of the first techniques to embed on-chain tokens in an extended version of Bitcoin was described in [13]. The technique relies on covenants, an extension of Bitcoin scripts which allows transactions to constrain the scripts of the redeeming ones. Roughly, a transaction output containing the script:

\[
e \text{ and } \text{verrec(rtxo}(n))
\]

where \(e\) is an arbitrary script, can only be redeemed by a transaction which makes \(e\) evaluate to true, and whose script in the \(n\)-th output is syntactically equal to \(e\) and \(\text{verrec}(\text{rtxo}(n))\).

Using covenants, we can mint a token by appending the transaction \(T\) below, where the extra field \(\text{arg}\) is syntactic sugar for a sequence of values accessible by the script:

\[
\text{out}(1): \{ \text{arg} : pk_A, \\
\text{scr} \triangleright \text{versig}(\text{ctxo}.arg, \text{rtx.wit}) \text{ and } \text{rtxo}(1).\text{val} = 1 \text{ and } \text{val} \triangleright 1\} \\
\]

The \(\text{arg}\) field identifies \(A\) as the owner of the token: to transfer the ownership to \(B\), \(A\) must spend \(T\) with a transaction \(T'\), setting its \(\text{arg}\) to \(B\)’s public key. For this to be possible, \(T'\) must satisfy the conditions specified in \(T\)’s script: (i) the \(\text{wit}\) field must contain the signature of the current owner; (ii) the output at index 1 must have \(1\) value, to preserve the value of the token; (iii) the script at index 1 in \(T'\) must be equal to that in \(T\). Once \(T'\) is on the blockchain, \(B\) can transfer the token to another user, by appending a transaction which redeems \(T'\).

Note that the transaction \(T\) above actually mints a non-fungible token, which can be transferred from one user to another, but whose value cannot be split (further, the token has a subtle flaw related to join actions: we will say more on this). The first step to turn the token into a fungible one is to support the split action. We can achieve this by adding a second element to the \(\text{arg}\) sequence, to represent the number of token units deposited in the transaction output. Using the notation \(w.i\) to access the \(i\)-th element of a sequence \(w\) (for \(1 \leq i \leq |w|\)), we can implement a splittable token as follows:

\[
\text{stxo}(n) \triangleq \text{output redeemed by the } n\text{-th input of rtx}
\]

The last two lines of the script ensure that any transaction which redeems \(T_{\text{split}}\) has exactly two outputs, each one with the same script of \(T_{\text{split}}\). The second line ensures that the \(\text{split}\) preserves the number of token units (here, \(\text{val}\) is immaterial).

Now, let \(\epsilon_{\text{split}}\) be the script used in \(T_{\text{split}}\). To extend the token with the join action, first we need to add a third element to the \(\text{arg}\) sequence, to encode the action performed by a transaction (say, \(G\) for \(\text{gen}\), \(S\) for \(\text{split}\), and \(J\) for \(\text{join}\)). The extended script could have the following form:

\[
e \triangleq \text{ if rtxo}(1).\text{arg}.3 = S \text{ then } \epsilon_{\text{split}} \text{ else } \epsilon_{\text{join}}
\]

where \(\epsilon_{\text{join}}\) implements the join functionality, i.e.: (i) verify the signature on the redeeming transaction; (ii) check that the redeeming transaction has exactly two inputs and one output; (iii) ensure that the token units are preserved; (iv) ensure that the joined transactions represent units of the same token.

For instance, consider the transactions in Figure 1, where \(T_4\) and \(T_3\) represent, respectively, the deposits \((B, 8 : \tau)_x\) and \((B, 2 : \tau)_x\). To perform the join action in (4), we must spend \(T_4\) and \(T_3\) with the transaction \(T_5\): this requires to satisfy the script \(e\) in \(T_4\) and \(T_3\). For condition (iii), the script must ensure that the 10 token units redeemed by \(T_5\) are the sum of the 8 units in \(T_4\) and the 2 units in \(T_3\). For condition (iv), the script in \(T_4\) should check that it is the same as that in \(T_3\), and vice versa. Hence, to implement conditions (iii)-(iv), the script in a transaction output must be able to access the fields in its sibling, i.e. the transaction output which is redeemed together (e.g., \((T_3, 1)\) is the sibling of \((T_4, 1)\) when appending \(T_5\)). However, neither Bitcoin nor its extensions with covenants [12]–[15] allow scripts to access the siblings.

An insecure implementation of join: To implement the join action, we start by extending Bitcoin scripts with an operator to access the sibling transaction outputs:

\[
\text{stxo}(n) \triangleq \text{output redeemed by the } n\text{-th input of rtx}
\]
Using this new operator, we can encode the conditions (iii) and (iv) in $e_{\text{join}}$ as follows:

$$rtxo(1).arg.2 = stxo(1).arg.2 + stxo(2).arg.2 \quad (iii)$$
$$\text{verrec}(stxo(1)) \text{ and } \text{verrec}(stxo(2)) \quad (iv)$$

Although this implementation of $e_{\text{join}}$ correctly encodes the conditions, it introduces a security vulnerability: an adversary can join two deposits of different tokens. The attack is exemplified in Figure 2. The transactions $T_A$ and $T_M$ mint 10 units of different tokens, and transaction $T_3$ joins them into a single deposit of the same token. Ideally, to counter this attack, $e_{\text{join}}$ should check not only the sibling, but also its ancestors until the minting transaction, and verify that it corresponds to the minting ancestor of the current transaction output. Although this would be possible by adding script operators that can go up the transaction graph at an arbitrary depth, this would be highly inefficient from the point of view of miners, who should record the whole transaction graph, instead of just the set of unspent transactions (UTXO).

**Neighbourhood covenants:** To address this issue, we use an operator which can go up the transaction graph only one level, i.e. up to the parent of the current transaction. Hence, implementing our Bitcoin extension with neighbourhood covenants requires miners to just record the UTXOs and their parents. By exploiting this new covenant, we can thwart the $\text{join}$ attack of Figure 2, and eventually obtain a secure and efficient implementation of fungible tokens. We now sketch the script $e_{\text{TOK}}$ which implements tokens. First, we add a fourth element to the $\text{arg}$ sequence, to record in each transaction output the identifier of the token deposited in that output. As an identifier, we use the hash of the parent of the minting transaction, which we access through the script $\text{txid}(ptxo(1))$. When we evaluate the script contained in the minting transaction (e.g., $T_A$ and $T_M$ in Figure 2), we require that its $\text{arg}$ field actually contains the token identifier: $e_{\text{gen}} \triangleq \cdots$ and $\text{ctxo}.arg.4 = \text{txid}(ptxo(1))$

Then, in the sub-scripts corresponding to all other token actions, we check that the redeeming transaction preserves the token identifier. E.g., in the $\text{join}$ sub-script, besides checking conditions (iii) and (iv) as shown before, we add the condition:

$$e_{\text{join}} \triangleq \cdots \text{ and } \text{ctxo}.arg.4 = rtxo(1).arg.4$$

In Figure 3 we show how this resolves the attack of Figure 2. In order to append the malicious $\text{join}$ transaction $T_3$, we must satisfy the script $e_{\text{TOK}}$ in both $T_1$ and $T_2$. These scripts check that the $\text{tokid}$ in the redeeming transaction $T_3$ is equal to the identifiers of the two branches, $H(T'_{A}, 1)$ and $H(T'_{M}, 1)$, by collision resistance of the hash function, this is not possible.

The discussion above shows how to counter an attack which attempts to $\text{join}$ different tokens. However, the adversary could devise more ingenious attacks, e.g. forging units of an existing token. For instance, if $T_M$ in Figure 3 were storing $\mathbb{B}$, its owner could spend it to create a transaction $T_2$ with arbitrary scripts and arguments. In particular, $T_2$ could use $e_{\text{TOK}}$ and any token identifier in $\text{arg.4}$, e.g., $H(T'_{A}, 1)$, effectively forging new units of a pre-existing token. Our full $e_{\text{TOK}}$ script exploits neighbourhood covenants to prevent these kinds of attacks as well (see e.g. the discussion before Lemma 4).

### III. A Symbolic Model of Tokens

Let $A, B, \ldots$ range over users, and let $\tau, \tau', \ldots$ range over tokens, encompassing both user-defined ones and bitcoins ($\mathbb{B}$). A term $\langle A, v : \tau \rangle_x$ represents a deposit of $v \in \mathbb{N}$ units of the token $\tau$ owned by $A$ (the index $x$ is an unique identifier of the deposit). A term $A \triangleright_x \alpha$ represents $A$’s authorization to perform the action $\alpha$ on the deposit $x$. The possible token actions are the following:

- **$\text{gen}(x, v)$** represents the act of spending a bitcoin deposit $x$ to mint $v$ units of a new token. The owner of these units is the user who owned the deposit $x$.
- **$\text{burn}(x, y)$** represents the act of destroying a sequence of deposits $x$, moving them to an unspendable deposit $y$.
- **$\text{split}(x, v, B)$** represents the act of splitting a deposit $x$ (say, containing $v + v'$ units of a token $\tau$) in two deposits of $\tau$. The first one of these deposits is owned by the same owner of $x$, and contains $v$ token units. The second one if owned by $B$, and contains the remaining $v'$ units.
- **$\text{join}(x, y, C)$** represents the joining of two deposits $x$ and $y$ of the same token, into a new deposit, owned by $C$. 

Fig. 2: A $\text{join}$ attack merging two different tokens.

Fig. 3: Thwarting the $\text{join}$ attack.
• \(xchg(x, y)\) represents the act of atomically exchanging the owners of the two deposits \(x\) and \(y\) (not both of bitcoins). In particular, when one of the deposits stores \(\mathbb{B}\), this action represents buying/selling tokens for bitcoins.

• \(\text{give}(x, B)\) represents a donation of the deposit \(x\) to \(B\).

Users follow a common pattern to perform token actions: (i) first, the involved users grant their authorization on the action; (ii) once all the needed authorizations have been granted, the action can actually be performed.

A configuration \(\Gamma\) is a compositions of deposits and authorizations. We assume that configurations form a commutative monoid under the composition operator \(|\cdot|\), and we use 0 to denote the empty configuration. We require that if \(\langle A, v : \tau \rangle_x\) and \(\langle B, v' : \tau' \rangle_x\) both occur in \(\Gamma\), then \(x \neq x'\). We define a transition semantics between configurations in Figure 4. Transitions are decorated with labels, which describe the performed actions. The rules for granting authorizations (Figure 8 in Appendix A) are straightforward.

Rule \([\text{GEN}]\) consumes a bitcoin deposit \(x\) owned by \(A\) to generate \(v\) units of a new token \(\tau\), which are stored in a fresh deposit \(y\). Note that \(A\)'s authorization is required to perform the action. For simplicity, we assume that minting tokens has no cost: it would be straightforward to adapt the rule to require a minting fee. Rule \([\text{BURN}]\) removes from the configuration a single token deposit (when \(n = 1\) and \(\tau_1 \neq \mathbb{B}\)), or atomically removes a sequence of bitcoin deposits (when \(\tau_i = \mathbb{B}\) for all \(i\)). Rule \([\text{SPLIT}]\) divides a deposit \(x\) in two fresh deposits \(y\) and \(z\), preserving the number of token units. Rule \([\text{JOIN}]\) allows \(A\) and \(B\) to merge two deposits of a token \(\tau\), preserving the amount of token units, and transferring the new deposit to \(C\). Rule \([\text{XCHG}]\) allows \(A\) and \(B\) to swap two deposits, containing either user-defined tokens or bitcoins. Finally, rule \([\text{Give}]\) allows \(A\) to donate one of her deposits to another user.

The transition relation \(\rightarrow\) is non-deterministic, because of the fresh names generated for deposits and tokens: however, given a transition \(\Gamma \xrightarrow{\alpha} \Gamma'\) the label \(\alpha\) is uniquely determined from \(\Gamma\) and \(\Gamma'\). A symbolic run \(S\) is a (possibly infinite) sequence \(\Gamma_0\Gamma_1\ldots\), where \(\Gamma_0\) contains only \(\mathbb{B}\) deposits, and for all \(i \geq 0\) there exists some (unique) \(\alpha_i\) such that \(\Gamma_i \xrightarrow{\alpha_i} \Gamma_{i+1}\). For all \(i \geq 0\), we denote with \(S_i\) the \(i\)-th element of the run, when this element exists. If \(S\) is finite, we denote its length as \(|S|\), and we write \(\Gamma_S\) for its last configuration, i.e. \(S_{|S|-1}\).

**Definition 1** (Token balance). We define the balance of a token \(\tau \neq \mathbb{B}\) in a configuration \(\Gamma\) inductively as follows:

\[
\begin{align*}
&\text{bal}_x(0) = 0 \\
&\text{bal}_x(\Gamma | \Gamma') = \text{bal}_x(\Gamma) + \text{bal}_x(\Gamma') \\
&\text{bal}_x(\langle A, v : \tau \rangle_x) = v \\
&\text{bal}_x(\langle A, v : \tau' \rangle_x) = 0 \quad (\tau' \neq \tau)
\end{align*}
\]

The following lemma establishes a basic preservation property: the balance of a token after a run is equal to the minted value minus the burnt value, defined as:

\[
\begin{align*}
\text{minted}_\tau(S) &= v \text{ if } \exists i : S_i \xrightarrow{\text{gen}(x, v)} S_{i+1}, \text{ and } \tau \text{ occurs in } S_{i+1} \text{ but not in } S_i \\
\text{burnt}_\tau(S) &= \sum \left\{ v \mid \exists i : S_i \xrightarrow{\text{burn}(x, y)} S_{i+1}, \text{ and } S_i = \Gamma | \langle A, v : \tau \rangle_x \right\}
\end{align*}
\]

**Lemma 1.** Let \(S\) be a finite symbolic run. For all \(\tau \neq \mathbb{B}\):

\[
\text{bal}_\tau(S) = \text{minted}_\tau(S) - \text{burnt}_\tau(S)
\]

**IV. Bitcoin Transactions**

In this section we recall the functionality of Bitcoin. To this purpose we rely on the formal model of [16], simplifying or omitting the parts that are irrelevant for our subsequent technical development (e.g., we abstract from the fact that, in the Bitcoin blockchain, transactions are grouped into blocks).

**Transactions:** Following the formalization in [16], we represent transactions as records with the following fields:\footnote{Bitcoin transactions can also impose time constraints on when they can be appended to the blockchain, or when they can be redeemed. Since time constraints are immaterial for our technical development, we omit them.}

• \(\text{in}\) is the list of inputs. Each of these inputs is a transaction output \(\langle T, i \rangle\), referring to the \(i\)-th output field of \(T\).

• \(\text{wit}\) is the list of witnesses, of the same length as \(\text{in}\).

Intuitively, for each \(\langle T, i \rangle\) in the \(\text{in}\) field, the witness at the same index must make the \(i\)-th output script of \(T\) evaluate to true.

• \(\text{out}\) is the list of outputs. Each output is a record of the form \{\text{scr}: e, \text{val}: v\}, where \(e\) is a script, and \(v \geq 0\).

We let \(f\) range over transaction fields, and we denote with \(T.f\) the content of field \(f\) of transaction \(T\). We write \(T.f(i)\) for the \(i\)-th element of the sequence \(T.f\), when in range; when the sequence has exactly one element, we write \(T.f\) for \(T.f(1)\). For transaction outputs \(\langle T, i \rangle\), we interchangeably use the notation \(\langle T, i \rangle\) and \(T.\text{out}(i)\), and we use the notation above to access their sub-fields \(\text{scr}\) and \(\text{val}\). When clear from the context, we just write the name \(A\) of a user in place of her public/private keys, e.g., we write \(\text{versig}(A, e)\) for \(\text{versig}(pk_A, e)\), and \(\text{sig}_A(T)\) for \(\text{sig}_{sk_A}(T)\).

**Scripts:** Bitcoin scripts are terms with the following syntax:

\[
\begin{align*}
&e ::= v \quad \text{constant (integer or bitstring)} \\
&| e \circ e \quad \text{operators } (\circ \in \{+, -, =, <\}) \\
&| e \text{ then } e \text{ else } e \quad \text{conditional} \\
&| e.n \quad n\text{-th element of sequence } e \ (n \in \mathbb{N}) \\
&| \text{rtx}, \text{wit} \quad \text{witnesses of the redeeming tx} \\
&| |e| \quad \text{size (number of bytes)} \\
&| H(e) \quad \text{hash} \\
&| \text{versig}(e, e') \quad \text{signature verification}
\end{align*}
\]

Besides constants \(v\), basic arithmetic/logical operators, and conditionals, scripts can access the elements of a sequence \((e.n)\), and the sequence of witnesses of the redeeming transaction \((\text{rtx}, \text{wit})\); further, they can compute the size \(|e|\) of a bitstring and its hash \(H(e)\). The script \(\text{versig}(e, e')\) evaluates to 1 if the signature resulting from the evaluation of \(e'\) is verified against the public key resulting from the evaluation of \(e\), and 0 otherwise.\footnote{Multi-signature verification is supported by Bitcoin scripts, but immaterial for our technical development: therefore, we omit it.} For all signatures, the signed message is the redeeming transaction (except its witnesses).

\[1\]
The semantics of scripts is in Figure 5. The script evaluation function $\texttt{eval}^T_i$ takes two additional parameters: $T$ is the redeeming transaction, and $i$ is the index of the redeeming input/witness. The result of the semantics can be an integer, a bitstring, or a sequence of integers/bitstrings. We denote with $H$ a public hash function, with $\texttt{size}(v)$ the size (in bytes) of an integer $v$, and with $\texttt{ver}$ a signature verification function (the definition of these semantic operators is standard, see e.g. [16]). The semantics of a script can be undefined, e.g. when accessing an element of a non-sequence: we denote this case as $\perp$. All the operators are $\textit{strict}$ i.e. they evaluate to $\perp$ if some of their operands is $\perp$. We use standard binary sugar for scripts, e.g.: (i) $\texttt{false} \triangleq 0$, (ii) $\texttt{true} \triangleq 1$, (iii) $e$ and $e'$ $\triangleq$ if $e$ then $e'$ else false, and finally (v) not $e$ $\triangleq$ if $e$ then false else true.

**Blockchains:** A blockchain $B$ is a finite sequence $T_0 \cdots T_n$, where $T_0$ is the only coinbase transaction (i.e., $T_{0.in} = \perp$). We say that the transaction output $(T_i,j)$ is spent in $B$ iff there exists some $T_{i'}$ in $B$ (with $i' > i$) and some $j'$ such that $T_{i'.in}(j') = (T_i,j)$. The $\texttt{unspent transaction outputs}$ of $B$, written $\texttt{UTXO}(B)$, is the set of transaction outputs $(T_i,j)$ which are unspent in $B$. A transaction $T$ is a $\textit{valid extension}$ of $B = T_0 \cdots T_n$ whenever the following conditions hold:

1) for each input $i$ of $T$, if $T.in(i) = (T',j)$ then:
   - $T' = T_h$, for some $h < n$ (i.e., $T'$ is in $B$);
   - the output $(T',j)$ is not spent in $B$;
   - $\texttt{[[T',j].scr]}_{T,i} = v \neq 0$;
2) the sum of the amounts of the inputs of $T$ is greater or equal to the sum of the amount of its outputs.

The Bitcoin consensus protocol ensures that each transaction $T_i$ in the blockchain is valid with respect to the sequence of past transactions $T_0 \cdots T_{i-1}$. The difference between the amount of inputs and that of outputs of transactions is the $\textit{fee}$ paid to miners who participate in the consensus protocol.

**V. Neighbourhood coveneants**

To extend Bitcoin with neighbourhood coveneants, we amend the model of pure Bitcoin in the previous section as follows:

- in transactions, we add a field to outputs, making them records of the form $\{\texttt{arg} : a, \texttt{scr} : e, \texttt{val} : v\}$, where $a$ is a sequence of values; Intuitively, this extra element can be used to encode a state within transactions;
- in scripts, we add operators to access all the outputs of the redeeming transaction, and a relevant subset of those of the sibling and parent transactions (by contrast, pure Bitcoin scripts can only access the redeeming transaction, and only as a whole, to verify it against a signature);
- in scripts, we add operators for coveneants.

We now formalize our Bitcoin extension. We use $o$ to refer to the following transaction outputs:

- $o ::= \texttt{rtxo}(e)$ output of the redeeming tx
- $| \texttt{stxo}(e)$ output of a sibling tx
- $| \texttt{ptxo}(e)$ output of a parent tx

More precisely, consider the case where a transaction output $(T',j)$ is redeemed by a transaction $T_i$, through its $i$-th input. When used within the script of $(T',j)$: $\texttt{rtxo}(n)$ refers to the $n$-th output of $T$; $\texttt{stxo}(o)$ refers to the output redeemed by the $n$-th input of $T$; $\texttt{ptxo}(n)$ refers to the output redeemed by the $n$-th input of $T'$. In Figure 6, we exemplify these outputs in relation to the transaction $T_i$. The semantics of $o$ is defined in the first line of Figure 7; its result is a pair $(T,n)$.

We extend scripts as follows, where $f \in \{\texttt{arg}, \texttt{val}\}$:

- $e ::= \cdots | f.o$ field of a tx output
- $\texttt{verscr}(e,o)$ basic coveneant
- $\texttt{verrec}(o)$ recursive coveneant
- $\texttt{inidx}$ index of redeeming tx input
- $\texttt{outidx}$ index of redeemed tx output
- $\texttt{inlen}(o)$ number of inputs
- $\texttt{outlen}(o)$ number of outputs
- $\texttt{txid}(o)$ hash of (tx,output)

The script $f.o$ gives access to the field $f$ of a transaction output $o$ (where $f$ is either $\texttt{arg}$ or $\texttt{val}$). The basic coveneant $\texttt{verscr}(e,o)$ checks that the script in the transaction output $o$ is syntactically equal to $e$ (note that $e$ is not evaluated). The “recursive” coveneant $\texttt{verrec}(o)$ checks that the script in $o$ is syntactically equal to script which is currently being evaluated. The operators $\texttt{inidx}$ and $\texttt{outidx}$ evaluate, respectively, to the index of the redeeming input and redeemed output. We call $\texttt{current transaction output}$ $(\texttt{ctxo})$ the transaction output which includes the script which is currently being evaluated, i.e.:

$\texttt{ctxo} \triangleq \texttt{stxo}(\texttt{inidx})$
\[ [v]_{T,i} = v \quad [e \circ e']_{T,i} = [e]_{T,i} \circ_* [e']_{T,i} \quad [\text{if } c_0 \text{ then } c_1 \text{ else } c_2]_{T,i} = \begin{cases} [c_0]_{T,i} \quad & \text{if } c_0 \\ [c_1]_{T,i} \quad & \text{if } c_1 \\ [c_2]_{T,i} \quad & \text{if } c_2 \end{cases} \]

\[ [e.a]_{T,i} = v_n \quad \text{if } [e]_{T,i} = v_1 \cdots v_k \quad (1 \leq n \leq k) \quad \text{if } \text{rtx.wit} = T \cdot \text{wit}(i) \]

\[ [\text{size}(e)]_{T,i} = \text{size}([e]_{T,i}) \quad [H(e)]_{T,i} = H([e]_{T,i}) \quad [\text{versig}(e, e')]_{T,i} = \text{versig}([e]_{T,i}, [e']_{T,i}, T, i) \]

Fig. 5: Semantics of Bitcoin scripts.

The scripts \( \text{inlen}(\cdot) \) and \( \text{outlen}(\cdot) \) evaluate, respectively, to the number of inputs and to the number of outputs of the transaction containing \( \cdot \). Finally, \( \text{tid}(\cdot) \) evaluates to a unique identifier of the transaction output \( \cdot \).

**Example 1.** Consider the transactions in Figure 6. The script in \( T_e \cdot \text{out}(1) \) checks that (i) the \( \text{arg} \) field of \( \text{ctxo} \), i.e. the same output which contains the script, equals to \( n_c \); (ii) the \( \text{arg} \) field of \( \text{ptx0}(1) \), i.e. the parent transaction output redeemed by \( T_e \cdot \text{in}(1) \), equals to \( n_p \); (iii) the \( \text{arg} \) field of \( \text{ctxo}(3) \), i.e. the third output of the redeeming transaction \( T_r \), equals to \( n_r \); (iv) the \( \text{arg} \) field of \( \text{ctxo}(2) \), i.e. the sibling transaction output redeemed by \( T_r \cdot \text{in}(2) \), equals to \( n_s \). Note that, in general, any script used in \( T_r \) can not access the parents of \( T_p \) and those of \( T_s \) — and in general all the transactions which are farther than those shown in the figure.

Figure 7 defines the semantics of extended scripts. As in Section IV, the function \( \lfloor \cdot \rfloor \) takes as parameters the re-deeming transaction \( T \) and the index \( i \) of the redeeming input. We denote with \( \equiv \) syntactic equality between two scripts, i.e. \( e \equiv e' \) is 1 when \( e \) and \( e' \) are exactly the same, 0 otherwise.

**Turing completeness:** Our neighbourhood covenants make Bitcoin Turing-complete. To prove this, we describe how to simulate in the extended Bitcoin any counter machine [17], a well-known Turing-complete computational model. A **counter machine** is a pair \((n, s)\), where \( n \in \mathbb{N} \) is the number of integer registers of the machine, and \( s \) is a sequence of instructions. Instructions are the following: \( \text{inc } i \) increments register \( i \), \( \text{dec } i \) decrements it, \( \text{zero } i \) sets it to zero, \( \text{if } i \neq 0 \text{ goto } j \) conditionally jumps to instruction \( j \) when register \( i \) is not zero, \( \text{halt } \) terminates the machine. The state of a counter machine \((n, s)\) is a tuple \((v_1, \ldots, v_n, p)\) where each \( v_i \) represents the current value of register \( i \), and \( p \) is the number of the next instruction to execute (i.e., the program counter). To exploit the currency transfer capabilities of Bitcoin, we slightly extend the counter machine model, by requiring that the machine has an initial \( \mathbb{B} \) balance which, upon termination, is transferred to the user \( A \) if the content of the first register is 0, or to \( B \) otherwise.

We call this extended model **UTXO-counter machine**.

**Theorem 2.** Neighbourhood covenants can simulate any UTXO-counter machine. Hence, they are Turing-complete.

**Proof.** We represent the state of the counter machine as a single transaction having one output. We simulate an execution step by appending a new transaction, which redeems the output representing the old state, and transfers its balance to a new output representing the new state. This is done until the machine halts, at which point we transfer its balance to the user determined by the final registers state. We remark that this simulation is made possible by the use of unbounded integers in our model, while Bitcoin only supports 32-bit integers.

We represent the machine state in the \( \text{arg} \) field of the transaction output \( o \), storing it as a sequence of integers. As a shorthand, we write \( o \cdot i \cdot r \) for \( o \cdot \text{arg}(i) \cdot r \) and \( o \cdot p \) for \( o \cdot \text{arg}(n+1) \). To simulate the execution steps of the machine, we define the script \( e_{CM} \), which checks that the new transaction \( T \) indeed represents the next state. More in detail, we first check whether the instruction in \( s \) position \( \text{ctxo} \cdot p \) is halt, in which case we require that \( T \) distributes the balance to users in the intended manner, depending on the current state. When \( \text{ctxo} \cdot p \) points to any other instruction, we start by requiring that \( T \) has only one output, the same balance, and the same script. We use a recursive covenant \( \text{verrec} \) on the redeeming transaction to ensure the last part. Then, we check that the new state in \( T \cdot \text{out}(1) \cdot \text{arg} \) agrees with the counter machine semantics, by cases on the instruction pointed to by \( o \cdot p \). If the instruction is \( \text{inc } i \), then we require \( \text{ctxo}(1) \cdot r_i = \text{ctxo} \cdot r_i + 1 \), \( \text{ctxo}(1) \cdot f_k = \text{ctxo} \cdot f_k \) for all \( k \neq i \), and \( \text{ctxo}(1) \cdot p = \text{ctxo} \cdot p + 1 \). The \( \text{dec } i \) and \( \text{zero } i \) cases are analogous. For the instruction if \( i \neq 0 \text{ goto } j \), we check the value of \( \text{ctxo} \cdot r_i \); if nonzero, we require that \( \text{ctxo}(1) \cdot p = j \), otherwise that \( \text{ctxo}(1) \cdot p = \text{ctxo} \cdot p + 1 \). In both cases, we require that \( \text{ctxo}(1) \cdot f_k = \text{ctxo} \cdot f_k \) for all \( k \).

Users start the simulation by appending to the blockchain a transaction \( T_0 \) having one output with the desired value, the script \( e_{CM} \), and a sequence of \( n + 1 \) zeros as \( \text{arg} \). After that, the balance is effectively locked inside the transaction, and the only way to transfer it back to the users is to simulate all the steps of the machine, until it halts. So, our simulated execution is a form of **secure multiparty computation** [18]–[20].

Although, for simplicity, we use UTXO-counter machines
just to transfer funds to A or B upon termination, it would be easy to generalise the computational model and simulation technique to encompass interactive computations, which at run-time can receive inputs and perform currency transfers. Doing so, we can execute arbitrary smart contracts, with the same expressiveness of Turing-complete smart contracts platforms. In principle, we could craft a smart contract which implements user-defined tokens in an account-based fashion: this contract would record the balance of the tokens of all users, and execute token actions. However, in practice this construction would be highly inefficient: performing a single token transfer would require to append a large number of transactions, and to pay the related fees.

To improve the efficiency of tokens, instead of limiting to covenants on the redeeming transaction, as in the simulation above, we could also use covenants on the parent and sibling transactions. In Section VI we show an efficient implementation of tokens, which fully exploits neighbourhood covenants.

Implementing neighbourhood covenants: To extend Bitcoin with neighborhood covenants, only small changes to the script language are needed. Currently, scripts can only access the redeeming transaction, and only for signature verification. To enable covenants, scripts need to access the fields of the redeeming transaction, and those of the parent and the sibling transaction outputs. First, the UTXO data structure must be extended to record the parents of unspent outputs. Full nodes could simply access these transactions from their identifiers. Lightweight nodes, i.e. Bitcoin clients with limited resources that store the UTXO instead of the whole blockchain, must also store parents beside the UTXO; when all the children of a transaction are spent, the parent can be deleted.

To implement the covenants verrec and verscr, the Bitcoin script language must be extended with new opcodes. A former covenant-enabling opcode is the CheckOutputVerify of [13], which uses placeholders to represent variable parts of the script (e.g., versig(<pubKey>, rtx.wit)). However, its implementation requires string substitutions at run-time to insert the wanted values in the script before checking script equality. Instead, our neighbourhood covenants can be implemented more efficiently. In our covenants, we can use exactly the same script along a chain of transactions, relying on the arg sequence to record state updates. The script can access the state through the operator o.arg, which require suitable opcodes. Since the script is fixed, nodes do not have to perform string substitutions, and checking script equality can be efficiently performed by comparing their hashes.

Adding the arg field to outputs does not require to alter the structure of pure Bitcoin transactions. Indeed, the arg values can be stored at the beginning of the script as push operations on the alternative stack, and copied to the main stack when the script refers to them, using the same technique used in [21]. When hashing the scripts for comparison, we discard these arg values: this just requires to skip the prefix of the script comprising all the push to the alternative stack.

VI. IMPLEMENTING TOKENS IN BITCOIN

In this section we show how to implement token actions in Bitcoin. To this purpose, we define a computational model, which describes the interactions of users who exchange messages and append transactions to the Bitcoin blockchain. A computational run C is a sequence of bitstrings γ, each of which encodes one of the following actions: (i) A → * : m; denoting the broadcast of a bitstring m; (ii) T, denoting the appending of a transaction T to the blockchain. A computational run always starts from a coinbase transaction T0. By extracting the transactions from a run C, we obtain a blockchain BC.

We relate the computational and the symbolic models through a relation between symbolic runs S and computational runs C: intuitively, S is coherent with C when each step in S is simulated by a step in C. In the rest of this section we provide the main intuition about the coherence relation, postponing the full details to Appendix B. The coherence relation, denoted as ∼, is parameterized over two injective functions txout and tkid which track, respectively, the names x and the tokens τ occurring in ΓS, mapping them to transaction outputs (T, i), where T occurs in C. We abbreviate S ∼txout.tkid C as S ∼ C.

We simulate each symbolic token action in Figure 4 by appending a suitable transaction to the blockchain. We use the arg part of transaction outputs to record the token data:

1. op is the action implemented by the transaction: 0 = gen, 1 = burn, 2 = split, 3 = join, 4 = xchg, 5 = give;
2. owner is the (public key of the) user who owns of the token units controlled by the tx output;
3. tkval is the number of units controlled by the tx output;
4. tkid is the unique token identifier.

We implement the token actions as a single script $\text{Cток}$, which uses a switch on the op value to jump to the first instruction of the requested action. Since the script is quite complex, we present independently the parts corresponding to each action, postponing the complete script to Appendix B (Figure 9). All the transactions implementing token actions use exactly the same script, which we preserve throughout executions by using recursive covenants. To improve readability,
we refer to the elements of the \texttt{arg} sequence by name rather than by index, e.g. we write \texttt{o.op} rather than \texttt{o.arg.1}.

**Gen:** We implement the \texttt{gen} action by the following script:

\begin{verbatim}
if not verrec(ptxo(1)) // ctxo is a gen
  then ctxo.tkid = tkid(ptxo(1)) // token id
  and ctxo.val = 0 // spent ctxo has 0 BTC
  and ctxo.val = 1 // gen has 1 out
  and ctxo.tval > 0 // positive token val
else ... // the other branches must preserve token id
\end{verbatim}

Recall that \texttt{gen} produces a symbolic step of the form:

$$\langle A, 0 : \emptyset \rangle \xrightarrow{\text{gen}(x,v)} \langle A, v : \tau \rangle_y$$

\((v > 0)\)

To translate this symbolic action into a computational one, we must spend a transaction output corresponding to \(\langle A, 0 : \emptyset \rangle_x\), and produce a fresh output corresponding to \(\langle A, v : \tau \rangle_y\).

Assuming that the deposit \(x\) corresponds to an unspent transaction output \((T',1)\) on the blockchain, this requires to append a transaction \(T\) redeeming \((T',1)\), and ensuring that:

1. the parent transaction output \((T',1)\) is not a token deposit, but just a plain \(\emptyset\) deposit (line 1);
2. \(tkid\) is the identifier of the parent tx output (line 2).

This corresponds to identifying the fresh name \(\tau\) with the deposit name \(x\) of the redeemed \(\emptyset\) deposit;
3. \((0,\emptyset)\) are redeemed from the parent transaction (line 3);
4. \(T\) has exactly one output (line 4);
5. \(tval\) is positive (line 5), corresponding to the constraint \(v > 0\) in the symbolic semantics.

Notice that the first time the script is evaluated is when redeeming \(T\) (not when appending it). At that time, \(ctxo\) will evaluate to \((T,1)\), and \(ptxo(1)\) to \((T',1)\). The script ensures that, when \(T\) is redeemed, its \(tkid\) will contain a unique identifier of the token. Crucially, the scripts which implement the other token actions will preserve this identifier in the \(tkid\) parameter. This identifier is essential to guarantee the correctness of the \texttt{join} and \texttt{xchg} actions.

**Burn:** We implement the \texttt{burn} action by the script:

\begin{verbatim}
versig(ctxo.owner, rtxo.wit) // check owner
and verrec(rtxo(1)) // covenants on rttx
and inlen(rtxo(1)) = 1 // rttx has 1 in
and outlen(rtxo(1)) = 1 // rttx has 1 out
\end{verbatim}

Recall that \texttt{burn} produces a symbolic step:

$$\langle A, (v + v') : \tau \rangle_x \xrightarrow{\text{burn}(x_1 \cdots x_n, y)} \langle C, (v + v') : \tau \rangle_z$$

There are two cases, according to whether we are burning a single token deposit, or one or more \(\emptyset\) deposits. In the first case, assuming that the computational counterpart of \(x_1\) is the output \((T',1)\), the corresponding computational step is to append a transaction \(T\) redeeming \((T',1)\). The witness of \(T\) must carry a signature of the owner \(A\), and its output script is \texttt{false}, making it unspendable. In the second case, it suffices to append a transaction \(T\) which redeems all the transaction outputs corresponding to \(x_1, \ldots, x_n\), and has a \texttt{false} script.

**Join:** We implement the \texttt{join} action by the script:

\begin{verbatim}
inlen(rtxo(1)) = 2 // rttx has 2 ins
and outlen(rtxo(1)) = 1 // rttx has 1 out
and verrec(rtxo(1)) // covenants on both inputs
and verrec(rtxo(2)) // covenants on both inputs
and rtxo(1).tkid = rtxo(1).tkid // preserve token id
and rtxo(1).tkid = rtxo(1).tkid // check sig of owner
and rtxo(1).tkid = rtxo(1).tkid // check sig of owner
\end{verbatim}

Recall that \texttt{join} produces a symbolic step of the form:

$$\langle A, v : \tau \rangle_x \| \langle B, v' : \tau \rangle_y \xrightarrow{\text{join}(x,y,C)} \langle C, (v + v') : \tau \rangle_z$$

Assume that \(x\) and \(y\) correspond to the unspent transaction outputs \((T',1)\) and \((T'',1)\), and ensuring that, for both inputs:

1. \(T\) has two inputs and one output, containing the same script of \((T',1)\) and \((T'',1)\) (line 1-6);
2. the token identifier \(tkid\) of the output of \(T'\) (rtxo(1)) is the same of \((T',1)\) (line 6);
3. the witness of \(T\) carries a signature of the owner (line 7);
4. the sum of token values \(tval\) of both inputs is equal to the token value of \((T',1)\) (line 8).

Note that the script in \((T',1)\) ensures that the one in \((T'',1)\) is the same, and vice-versa. In this way, we prevent joining tokens with bitcoins. To also prevent joining tokens of different type, the script checks that the \(tkid\) of the current transaction is the same as the one of the first output of the redeeming transaction. This is done by both inputs. In other words, \(stxo(1).tkid = rtxo(1).tkid\) and \(stxo(2).tkid = rtxo(1).tkid\), implying that \(stxo(1).tkid = stxo(2).tkid\).
Exchange: We implement the \( xchg \) action by the script:

\[
\begin{align*}
&\text{inlen}(\text{rtxo}(1)) = 2 \quad // \text{rtxo has 2 ins} \\
&\text{and outlen}(\text{rtxo}(1)) = 2 \quad // \text{rtxo has 2 outs} \\
&\text{and verrec(\text{stxo}(1)))} \quad // \text{covenant on input 1} \\
&\text{and versig(\text{ctxo.owner, rtx.wit})} \quad // \text{check owner} \\
&\text{and rtxo(1).owner = stxo(2).owner} \quad // \text{exchange owner} \\
&\text{and rtxo(2).owner = stxo(1).owner} \quad // \text{preserve value} \\
&\text{and rtxo(1).tkid = stxo(1).tkid} \quad // \text{preserve tkid} \\
\end{align*}
\]

if \( \text{verrec(stxo(2))} \) then // exchange token/token

\[
\begin{align*}
&\text{verrec(\text{rtxo}(2)))} \quad // \text{covenant on rtxo(2)} \\
&\text{and rtxo(2).tkval = stxo(2).tkval} \quad // \text{preserve tkval} \\
&\text{and rtxo(2).tkid = stxo(2).tkid} \quad // \text{preserve tkid} \\
\end{align*}
\]

else // exchange token/BTC

\[
\begin{align*}
&\text{verscr(versig(\text{ctxo.owner, rtx.wit}, \text{rtxo}(2)))} \\
&\text{and rtxo(2).val = stxo(2).val} \quad // \text{preserve BTC}
\end{align*}
\]

Recall that the symbolic \( xchg \) step has the form:

\[
\langle A, v : \tau \rangle_x \langle B, v' : \tau' \rangle_y \xrightarrow{xchg(x,y)} \langle A', v' : \tau' \rangle_{y'} \langle B, v : \tau \rangle_{y'}
\]

where \( \tau \) must be a token, while \( \tau' \) is either a \( \emptyset \) or a token.

Assume that \( x \) and \( y \) correspond to the unspent transaction outputs \((T', 1)\) and \((T'', 1)\). To perform the corresponding computational step we append a transaction \(T\) redeeming \((T', 1)\) and \((T'', 1)\), and ensuring that, for both inputs:

1. \(T\) has two inputs and two outputs (lines 1-2);
2. the first input and the first output of \(T\) must contain the same script of \((T', 1)\) and \((T'', 1)\) (lines 3-4);
3. the witness of \(T\) carries a signature of the owner (line 5);
4. the owner in the first output of \(T\) (rtxo(1)) must be equal to the owner in the second input of \((T', 1)\) (line 6);
5. dually, the owner in the second output of \(T\) (rtxo(2)) must be equal to the owner in the first input of \((T', 1)\) (line 7);
6. the token value and identifier of the first output of \(T\) (rtxo(1)) must be equal to those of \((T', 1)\) (line 8-9).

Furthermore, if \( \text{verrec(stxo(2))} \) is true, i.e. we are exchanging a token with a token. In this case, we require that:

1. the second output of \(T\) must contain the same script of \((T', 1)\) and \((T'', 1)\) (line 11);
2. the token value and identifier of the second output of \(T\) (rtxo(2)) must be equal to those of \((T', 1)\) (line 12-13).

When exchanging a token with a \( \emptyset \) deposit, we require:

1. the script of the second output of \(T\) (rtxo(2)) to be \( \text{versig(\text{ctxo.owner, rtx.wit})} \) (line 15);
2. the value of the second output of \(T\) to be equal to the value of \((T'', 1)\) (line 16).

Give: We implement the \( \text{give} \) action by the script:

\[
\begin{align*}
&\text{inlen}(\text{rtxo}(1)) = 1 \quad // \text{rtxo has 1 ins} \\
&\text{and outlen}(\text{rtxo}(1)) = 1 \quad // \text{rtxo has 1 outs} \\
&\text{and versig(\text{ctxo.owner, rtx.wit})} \quad // \text{check owner} \\
&\text{and verrec(\text{rtxo}(1)))} \quad // \text{covenant on rtxo(1)} \\
&\text{and rtxo(1).tkid = ctxo.tkid} \quad // \text{preserve tkid} \\
&\text{and rtxo(1).tkval = ctxo.tkval} \quad // \text{preserve tkval}
\end{align*}
\]

Recall that \( \text{give} \) produces a symbolic step of the form:

\[
\langle A, v : \tau \rangle_x \xrightarrow{\text{give}(x,B)} \langle B, v : \tau \rangle_y
\]

Assuming that \( x \) corresponds to an unspent transaction output \((T', 1)\), performing this step in Bitcoin requires to append a transaction \(T\) redeeming \((T', 1)\), and ensuring that:

1. \(T\) has only one input and one output (lines 1-2);
2. the witness of \(T\) carries a signature of the owner (line 3);
3. the output of \(T\) (rtxo(1)) contains the same script of \((T', 1)\) (line 4);
4. the token value and identifier of the first output of \(T\) (rtxo(1)) is the same as those of \((T', 1)\) (lines 5-6).

Authorizations: Symbolic authorization steps (see Figure 8 in Appendix A) correspond to computational broadcasts of signatures. Computational users, however, can also broadcast other messages: in particular, adversaries can broadcast any arbitrary bitstring they can compute in PPTIME. Coherence discards any broadcast which does not correspond to any of the symbolic steps above, i.e., such broadcasts correspond to no symbolic steps. Discarding these messages does not affect the security of tokens, because the other computational messages (i.e., transactions and their signatures) are enough to reconstruct the symbolic run from the computational one.

Other transactions: A subtle case of coherence is that of transactions appended by dishonest users. To illustrate the issue, suppose that some dishonest \( A_1, \ldots, A_n \) own some bitcoins, represented in the symbolic run as deposits \( \langle A_i, v_i : \emptyset \rangle_{x_i}, \) for \( i \in 1..n \), and in the computational run as transaction outputs \( T_{x_1} \cdots T_{x_n} \) each one redeemable with \( A_i \)'s private key. The dishonest users can sign an arbitrary \( T' \) which redeems all the \( T_{x_1} \cdots T_{x_n} \), append it to the blockchain. Crucially, \( T' \) may not correspond to \( \text{gen}, \text{split}, \text{join}, \text{xchg} \) or \( \text{give} \) actions. In this case, to obtain coherence, we simulate the appending of \( T' \) with a (previously authorized) \( \text{burn} \) of the deposits \( \langle A_i, v_i : \emptyset \rangle_{x_i} \). In subsequent steps, coherence will ignore the descendants of \( T' \) in the computational run, since in general they cannot be represented symbolically. More precisely, appending a transaction where none of the inputs corresponds to a symbolic deposit results in no symbolic action. The case of a deposit \( \langle A, v : \tau \rangle_x \) with a user-generated token \( \tau \) is similar: appending \( T' \) is simulated by a symbolic \( \text{burn} \), and \( T' \) is not represented symbolically. In all cases, a transaction which spends a symbolically-represented input must be represented symbolically, otherwise coherence is lost.

Efficiency of the implementation: To estimate the efficiency of the implementation, we consider the number of cryptographic operations, as their execution cost is an order of magnitude greater than the other operations. In particular, performing \( \text{verrec} \) and \( \text{verscr} \) requires to compute the hash of a script (once this is done, the cost of comparing two hashes is negligible). This cost can be reduced by incentivizing nodes to cache scripts. The most expensive token action is \( xchg \), which, having two inputs, needs to verify 2 signatures and execute at most 10 covenants operations, which overall require to compute at most 6 script hashes. If nodes cache scripts, the cost of the action is not dissimilar to the one required to append a standard transaction with two inputs. Note that, even though \( e\text{TOK} \) is a non-standard script, it could be used in a standard P2SH transaction, as in [21].
if it did not exceed the 520-bytes limit. Taproot [22] would mitigate this issue: for scripts with multiple disjoint branches, Taproot allows the witness of the redeeming transaction to reveal just the needed branch. Therefore, the 520-bytes limit would apply to branches instead of the whole script.

VII. Computational soundness

An immediate consequence of the definition of coherence is the following lemma, which ensures that unspent deposits in a symbolic run \( S \) have a corresponding unspent transaction output in any computational run \( C \) coherent with \( S \).

**Lemma 3.** Let \( S \sim C \). For each deposit \( \langle A, v : \tau \rangle_x \) in \( \Gamma_S \), there exists a distinct tx output \( txout(x) \) in UTXO(\( B_C \)) storing \( v : \tau \), and spendable through a signature of \( A \).

**Lemma 4** is a sort of dual of Lemma 3, describing how to relate computational token deposits to symbolic ones. In general, this is complex since a computational adversary can craft arbitrary scripts which are not representable in the symbolic model; hence, the blockchain can contain transactions \( T \) which do not correspond to any symbolic deposit. A tricky case is when, after some token \( \tau \) is minted in the symbolic world, a computational adversary creates a descendent \( T' \) of \( T \) with the token script and \( tkid = \tau \), effectively forging new units of \( \tau \). Note that such forgery can not be prevented, since the adversary can create transactions with arbitrary outputs. However, such forged tokens are useless: this is guaranteed by Lemma 4, which shows that \( T' \) is unspendable.

**Lemma 4.** Let \( S \sim C \). Let \( (T, i) \) be a tx output in UTXO(\( B_C \)) storing \( v : \tau \), with \( \tau \neq \emptyset \). If \( (T, i).tkid \) does not correspond to any tx output in \( B_C \), then \( (T, i) \) is unspendable. Otherwise, if \( (T, i).tkid \) occurs in \( B_C \), and \( (T, i).tkid = equal to txout(x) \) for some \( x \) such that \( gen(x, v') \) is fired in \( S \), then:

1) if \( (T, i) \notin \text{ran}(txout) \) then \( (T, i) \) is unspendable;
2) if \( (T, i) = txout(y) \) for some \( y \), then \( (T, i) \) is spendable, and \( \langle A, v : \tau \rangle_y \) occurs in \( \Gamma_S \), where \( A = (T, i).owner \).

Note that all the results above require as hypothesis that \( S \sim C \). In the rest of this section we show that, with overwhelming probability, for each computational run \( C \) there exists a symbolic run \( S \) which is coherent with \( C \). Actually, our computational soundness result is more precise than that. First, we consider only a subset \( \text{Hon} \) of users to be honest, while we consider all the others dishonest: without loss of generality, we model them as a single adversary \( \text{Adv} \). We assume that both honest users and the adversary have a strategy, which allows them to decide the next actions, according to the past run. Our computational soundness result establishes that, with overwhelming probability, any computational run conforming to the (computational) strategies has a corresponding symbolic run conforming to the corresponding (symbolic) strategies. Consequently, if there exists an attack at the computational level, then the attack is also observable at the symbolic level.

**Symbolic strategies:** The strategy \( \Sigma_A^S \) of a honest user \( A \) is a PTIME algorithm which allows \( A \) to select which action(s) to perform, among those permitted by the token semantics. \( \Sigma_A^S \) receives as input a finite symbolic run \( S \), and outputs a finite set of enabled actions, with the constraint that \( \Sigma_A^S \) cannot output authorizations for \( B \neq A \). We require strategies to be persistent: if on a run \( \Sigma_A^S \) chooses an action \( \alpha \), and \( \alpha \) is not taken as the next step in the run (e.g., because some other user acts earlier), then \( \Sigma_A^S \) must still choose \( \alpha \) after that step, if still enabled. The adversary \( \text{Adv} \) acts on behalf of all the dishonest users, and controls the scheduling among all users (including the honest ones). Her symbolic strategy \( \Sigma_{\text{Adv}}^S \) is a PTIME algorithm taking as input the current run and the sets of moves outputted by the strategies of honest users. The output of \( \Sigma_{\text{Adv}}^S \) is a single action \( \alpha \) (to be appended to the current run). To rule out authorization forgeries, we require that if \( \alpha \) is an authorization by some honest \( A \), then it must be chosen by \( \Sigma_A^S \). Fixing a set of strategies \( \Sigma^S \) — both for the honest users and for the adversary — we obtain a unique run, which is made by the sequence of actions chosen by \( \Sigma_{\text{Adv}}^S \) when taking as input the outputs of the honest users’ strategies. We say that this run is conformant to \( \Sigma^S \). When \( \Sigma_{\text{Adv}}^S \notin \Sigma^S \), we say that \( S \) conforms to \( \Sigma^S \) when there exists some \( \Sigma_{\text{Adv}}^S \) such that \( S \) conforms to \( \Sigma^S \cup \{ \Sigma_{\text{Adv}}^S \} \).

**Computational soundness:** We are now ready to establish our main result. We assume that cryptographic primitives are secure, i.e., hashes are collision resistant and signatures cannot be forged (except with negligible probability).

**Theorem 5.** Let \( \Sigma^S \) be a set of symbolic strategies for all \( A \in \text{Hon} \). Let \( \Sigma^C \) be a set of computational strategies such that \( \Sigma^C_A = \kappa(\Sigma^S_A) \) for all \( A \in \text{Hon} \), and including...
an arbitrary adversary strategy $\Sigma_{k_0}$. For each $k \in \mathbb{N}$ and security parameter $\eta$, we define the following experiment:

1) generate $C$ conforming to $\Sigma^c$, with $|C| \leq \eta^k$;
2) if there exists $S$ conforming to $\Sigma^s$ such that $S \sim C$ then return 1, otherwise return 0.

Then, the experiment returns 1 with overwhelming probability w.r.t. the security parameter $\eta$.

As an application of our results, we lift the balance preservation result of Lemma 1 from the symbolic to the computational model. We start by defining the balance of a computational token. Then, in Theorem 6 we establish the equivalence between symbolic token balance and the computational one.

**Definition 2** (Balance of a computational token). For all computational runs $C$ and transaction outputs $t$, let:

$$P_t = \{t' \mid t' \in UTXO(B_C) \text{ and } t' \text{ is spendable and } t'.tkid = t \land t'.scr = e_{TOK} \}$$

We define the balance of the computational token $t$ in $C$ as:

$$bal_t(C) = \sum_{t' \in P_t} t'.tkval$$

**Theorem 6.** Let $S \sim C$. If $\text{gen}(x, v)$ is fired in $S$, generating a fresh token $\tau$, then:

$$bal_{\tau}(S) = bal_{txout(x)}(C)$$

**Proof.** We first prove that $bal_{\tau}(S) \leq bal_{txout(x)}(C)$. For each $(A, v : \tau) \in \Gamma_S$, by Lemma 3 there exists a distinct spendable $(T, i) = txout(y)$ in $UTXO(B_C)$ such that $(T, i)\.owner = A$, $(T, i).scr = e_{TOK}$, $(T, i).tkid = txout(x)$. Then, $(T, i) \in P_{txout(x)}$, and so $v$ contributes to $bal_{txout(x)}(C)$. We now prove that $bal_{\tau}(S) \geq bal_{txout(x)}(C)$. Let $t \in P_{txout(x)}$. Since $t \in UTXO(B_C)$ and it is spendable, by item 2 of Lemma 4 some deposit $(A, v : \tau)$ occurs in $\Gamma_S$, with $A = t\.owner$. Then, $v$ contributes to $bal_{\tau}(C)$. \(\square\)

**VIII. RELATED WORK AND CONCLUSIONS**

We have proposed a secure and efficient implementation of fungible tokens on Bitcoin, exploiting neighbourhood covenants, a powerful yet simple extension of the Bitcoin script language. The security of our construction is established as a computational soundness result (Theorem 5). This guarantees that adversaries at the Bitcoin level cannot make tokens diverge from the ideal behaviour specified by the symbolic token semantics, unless with negligible probability.

To keep the presentation simple, we have slightly limited the functionality of tokens, making split/join/exchange actions operate on just two deposits, and omitting time constraints. Lifting these restrictions would only affect the size, but not the complexity, of our technical development. Further, it would allow to use tokens as within high-level languages for Bitcoin contracts, e.g. BitML [23], [24], simplifying the design of financial contracts which manage tokens. For instance, we could model as follows a basic zero-coupon bond [25] where an investor $A$ pays upfront to a bank $B$ 5 units of a token $\tau$, and receives back 1 $\$B$ after a maturity date $t$:

$$\text{split}(5: \tau \rightarrow \text{withdraw } B | 1: \$B \rightarrow \text{after } t: \text{withdraw } A)$$

A research question arising from our work is how to exploit neighbourhood covenants in the design of high-level languages for Bitcoin contracts. Besides enhancing the expressiveness of these languages (e.g., by allowing for unbounded recursion), neighbourhood covenants would enable a simpler compilation technique, compared e.g. that used in BitML. Indeed, to guarantee the liveness of contracts, BitML requires the participants in a contract to pre-exchange their signatures on all the transactions obtained by the compiler. By using covenants, we could avoid this overhead, since the logic which controls that a contract action is permitted is not based on signatures, but is implemented by the covenant.

**Related work:** In Ethereum, similarly to our approach, tokens are implemented on top of the platform using a smart contract, following the ERC-20 and ERC-712 standards [26], [27]. An alternative approach is to provide a native support for tokens: the works [28], [29] follow this approach, proposing an extension of the UTXO model used in the Cardano blockchain [30]. The work [31] also proposes an extension of the UTXO model that supports native tokens. Unlike the previous approaches, in this model there is no privileged cryptocurrency: transaction fees are paid in the same currency which is being exchanged. Algorand [9] supports native tokens in an account-based model, allowing their minting, burn, transfer, and their exchange through atomic groups of transactions.

The first proposals of covenants in Bitcoin date back at least to 2013 [32]. Nevertheless, their inclusion into the official Bitcoin protocol is still uncertain, mainly because of the cautious approach of the Bitcoin community to accept extensions [10]. The emerging of Bitcoin layer-2 protocols like the Lightning Network [33] has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [11]). Currently, covenants are supported by Bitcoin Cash [34], a mainstream blockchain platform originated from a fork of Bitcoin.

The work [13] proposes a new opcode CheckOutputVerify to explicitly constrain the outputs of the redeeming transaction, while [15] implements covenants by extending the current implementation of versig with a new opcode CheckSigFromStack, which can check a signature on arbitrary data on the stack. Both [13] and [15] can constrain the script of the redeeming transaction to contain the same covenant of the spent one, enabling recursive covenants similarly to our verrec$(txout(n))$.

An alternative approach is to implement covenants without adding new opcodes. This approach is proposed by [12], which relies on modified signature verification scheme, allowing users to sign a transaction template, i.e. to sign only parts of a transaction, leaving the state parameters variable. The idea that covenants would allow to implement state machines on Bitcoin was first made by [15]: in Theorem 2 we show...
that, using our neighbourhood covenants, Bitcoin can actually simulate counter machines, assuming unbounded integers.

The work [35] describes an approach to extend the UTXO model with a variant of covenants which, similarly to ours, can access the redeeming and the sibling transactions. Besides this, the extension in [35] features registers, loops, complex data structures, and native tokens, making it quite distant from an implementation on Bitcoin. Compared to [35], our extension devises a minimal extension to the UTXO model, so to allow for an efficient implementation on Bitcoin, and support secure fungible tokens.

The work [14] introduces a formal model of covenants, which can be implemented in Bitcoin with modifications similar to the ones discussed in Section V. However, since the covenants in [14] only constrain the redeeming transaction, its implementation is simpler, since it does not require to extend the UTXO set with the parents. As a downside, the covenants of [14] do not allow to implement fungible tokens.

REFERENCES

[1] “Ethereum token dynamics,” https://stat.bloxy.info/superset/dashboard/tokens.
[2] M. D. Angelo and G. Salzer, “Tokens, types, and standards: Identification and utilization in Ethereum,” in DAPPS. IEEE, 2020, pp. 1–10.
[3] M. Fröwis, A. Fuchs, and R. Böhme, “Detecting token systems on Ethereum,” in Financial Cryptography and Data Security, ser. LNCS, vol. 11598. Springer, 2019, pp. 93–112.
[4] “Epgbc protocol specification,” 2020, https://github.com/chromaway/ngeccbase/wiki/EPGBCSimple.
[5] “Openassets protocol,” 2020, https://github.com/OpenAssets.
[6] “Colo website,” 2020, https://www.colo.com/.
[7] “Counterparty website,” 2020, http://counterparty.io/.
[8] M. Bartoletti, B. Bellomy, and L. Pompiann, “A journey into Bitcoin metadata,” J. Grid Comput., vol. 17, no. 1, pp. 3–22, 2019.
[9] “Algorand developer docs - assets,” 2020, https://developer.algorand.org/docs/features/asa/.
[10] L. Dashj, “BIP 0002,” 2016, https://en.bitcoin.it/wiki/BIP_0002.
[11] J. Rubin, “CHECKTEMPLATEVERIFY,” 2020, BIP 119, https://github.com/bitcoin/bips/blob/master/bip-0119 mediawiki.
[12] J. Swanbo, S. Hommel, B. McElrath, and B. Bishop, “Bitcoin covenants: Three ways to control the future,” CoRR, vol. abs/2006.16714, 2020.
[13] M. Möser, I. Eyal, and E. G. Sirer, “Bitcoin covenants,” in Financial Cryptography Workshops, ser. LNCS, vol. 9604. Springer, 2016, pp. 126–141.
[14] M. Bartoletti, S. Lande, and R. Zunino, “Bitcoin covenants unchained,” in ISoLA, 2020, to appear.
[15] R. O’Connor and M. Piekarska, “Enhancing Bitcoin transactions with covenants,” in Financial Cryptography Workshops, ser. LNCS, vol. 10323. Springer, 2017.
[16] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, “A formal model of Bitcoin transactions,” in Financial Cryptography and Data Security, ser. LNCS, vol. 10937. Springer, 2018.
[17] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg, “Counter machines and counter languages,” Mathematical systems theory, vol. 2, no. 3, pp. 265–283, 1968.
[18] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, “Secure multiparty computations on Bitcoin,” Commun. ACM, vol. 59, no. 4, pp. 76–84, 2016.
[19] A. C. Yao, “How to generate and exchange secrets (extended abstract),” in 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986. IEEE Computer Society, 1986, pp. 162–167.
[20] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game or A completeness theorem for protocols with honest majority,” in ACM Symposium on Theory of Computing. ACM, 1987, pp. 218–229.
[21] “Balzac: Bitcoin abstract language, analyzer and compiler,” https://blockchain.unica.it/balzac/, 2018.

[22] A. T. Pieter Wuille, Jonas Nick, “Taproot: Segment less script for security, efficiency, and privacy,” in LNSC, 2020. 
[23] M. Bartoletti and R. Zunino, “BitML: a calculus for Bitcoin smart contracts,” in ACM CCS, 2018.
[24] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, “Developing secure Bitcoin contracts with BitML,” in ESEC/FSE, 2018.
[25] S. L. P. Jones, J. Eber, and J. Seward, “Composing contracts: an adventure in financial engineering, functional pearl,” in International Conference on Functional Programming (ICFP), 2000, pp. 280–292.
[26] “ERC-20 token standard,” 2015, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.
[27] W. Entringer, D. Shirley, J. Evans, and N. Sachs, “EIP 721: ERC-721 non-fungible token standard,” https://eips.ethereum.org/EIPS/eip-721.
[28] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. P. Jones, P. Vinogradova, P. Wadler, and J. Zahnentferner, “UTXOaug: UTXO with multi-asset support,” in ISoLA, 2020, to appear.
[29] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. P. Jones, P. Vinogradova, and P. Wadler, “Native custom tokens in the extended UTXO model,” in ISoLA, 2020, to appear.
[30] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler, “The extended UTXO model,” in Workshop on Trusted Smart Contracts, 2020.
[31] J. Zahnentferner, “Multi-currency ledgers,” Cryptology ePrint Archive, Report 2020/895, 2020, https://eprint.iacr.org/2020/895.
[32] G. Maxwell, “Cointokens: discounting SCIP signatures, an amusingly bad idea,” 2013, https://bitcoin-talk.org/index.php?topic=278122.0.
[33] J. Poon and T. Dryja, “The Bitcoin Lightning Network: Scalable off-chain instant payments,” 2015. [Online]. Available: https://lightning.network/lightning-network-paper.pdf.
[34] R. Kalso, “Cashscript — writing covenants,” 2019. [Online]. Available: https://cashscript.org/docs/guides/covenants/.
[35] A. Chepurnoy and A. Saxena, “Multi-stage contracts in the UTXO model,” in Cryptocurrencies and Blockchain Technology, ser. LNCS, vol. 11737. Springer, 2019, pp. 244–254.
APPENDIX A
SUPPLEMENTARY MATERIAL FOR SECTION III

Proof of Lemma 1
By induction on \( S \), and by case analysis on each step. Inspecting each symbolic semantics rule, we can see that each step preserves the amount of token units, except for minting ((\texttt{GEN}) and burning ((\texttt{BURN})), which are explicitly taken into account by the equation.

APPENDIX B
SUPPLEMENTARY MATERIAL FOR SECTION VI
We now formalize the coherence relation between symbolic and computational runs sketched in Section VII. Below, we call \( e_{\text{TOK}} \) the token script in Figure 9, and as \( e_{\text{BTC}} \) the script:

\[
\text{versig(ctxo.owner, rtx.wit)}
\]

We inductively define the relation:

\[
\text{coher}(S, C, txout, tkid, txburn)
\]

where:
- \( S \) is a symbolic run;
- \( C \) is a computational run;
- \( txout \) is an injective function, mapping the names \( x \) occurring in \( \Gamma_S \) (except those only in authorizations) to unspent transaction outputs \( (T, i) \) (where \( T \) occurs in \( C \));
- \( tkid \) is an injective function, mapping the tokens \( \tau \) occurring in \( \Gamma_S \) to unspent transaction outputs \( (T, i) \) (where \( T \) occurs in \( C \));
- \( txburn \) is an injective function, mapping the names \( x \) occurring in \( \Gamma_S \) only in \text{burn} authorizations to off-chain transactions.

We require the relation to respect the following invariants:
- if \( txout(x) = (T, i) \), then for all \( (T', i) \in T.in \), there exists \( y \) such that \( txout(y) = (T', i) \).
- if \( \Gamma_S = \{ A, v : \mathcal{B} \} \) then there exist \( T, i \) such that \( txout(x) = (T, i) \) and:
  1) \( (T, i).\text{arg} = p_kA \)
  2) \( (T, i).\text{scr} = \text{versig(ctxo.arg, rtx.wit)} \)
  3) \( (T, i).\text{val} = v \).
- if \( \Gamma_S = \{ A, v : \tau \} \) with \( \tau \neq \mathcal{B} \) then there exist \( T, i, T', j \) such that \( txout(x) = (T, i), tkid(\tau) = (T', j) \), and:
  1) \( (T, i).\text{op} = 0 \)
  2) \( (T, i).\text{tkid} = (T', j) \)
  3) \( (T, i).\text{owner} = A \)
  4) \( (T, i).\text{tkval} = v \)
  5) \( (T, i).\text{scr} = e_{\text{TOK}} \)
  6) \( (T, i).\text{val} = 0 \).

Base case: \( \text{coher}(S, C, txout, tkid, txburn) \) holds if the following conditions hold:
- \( S \) and \( C \) are initial, i.e.: \( S \) is initial if it contains only bitcoin deposits, and \( C \) only contains from a coinbase transaction \( T_0 \);
- \( txout \) injectively maps exactly the names \( x \) of the deposits \( \{ A, v : \mathcal{B} \} \) in \( \Gamma_S \) to outputs \( (T_0, i) \) of the form:
  1) \( (T_0, i).\text{arg} = p_kA \)
  2) \( (T_0, i).\text{scr} = \text{versig(ctxo.arg, rtx.wit)} \)
  3) \( (T_0, i).\text{val} = v \)
  4) \( \text{dom} (tkid) = \emptyset \)

Inductive case 1: \( \text{coher}(S, C, txout, tkid, txburn) \) holds if \( \text{coher}(S, C, txout', tkid', txburn') \) and one of the following cases applies:

1) \( \alpha = \text{gen}(x, v) \). In \( \Gamma_S \) there exists a deposit \( \{ A, 0 : \mathcal{B} \} \) and \( \gamma = T \), where:
   - \( T.in = txout(x) \)
   - \( T \) has exactly one output
   - \( (T, 1).\text{op} = 0 \)
   - \( (T, 1).\text{tkid} = txout'(x) \)
   - \( (T, 1).\text{owner} = A \)
   - \( (T, 1).\text{tkval} = v \)
   - \( (T, 1).\text{scr} = e_{\text{TOK}} \)
   - \( (T, 1).\text{val} = 0 \)

   In \( \Gamma_{S_0} \), there exists a deposit \( \{ A, v : \tau \} \) which did not exist in \( \Gamma_S \). The mapping \( txout \) inherits all the bindings of \( txout' \) except that for \( x \), and extends it with the binding \( y \mapsto (T, 1) \). The mapping \( tkid \) extends \( tkid' \) with the binding \( \tau \mapsto txout'(x) \). The mapping \( txburn \) is equal to \( txburn' \).

2) \( \alpha = \text{split}(x, v, \mathcal{B}) \). In \( \Gamma_S \) there exists a deposit \( \{ A, (v + v') : \tau \} \) and \( \gamma = T \). The following conditions hold:
   - \( T.in = txout'(x) \)
   - \( T \) has exactly two outputs (of indices 1 and 2)
   - \( (T, 1).\text{owner} = A \)
   - \( (T, 2).\text{owner} = B \)

   Further, if \( \tau = \mathcal{B} \), the following conditions hold:
   - \( (T, 1).\text{scr} = (T, 2).\text{scr} = e_{\text{BTC}} \)
   - \( (T, 1).\text{val} = v \)
   - \( (T, 2).\text{val} = v' \)

   Further, if \( \tau \neq \mathcal{B} \), the following conditions hold:
   - \( (T, 1).\text{op} = 2 \)
   - \( (T, 1).\text{tkid} = (T, 2).\text{tkid} = txout'(x).\text{tkid} \)
   - \( (T, 1).\text{tkval} = v \)
   - \( (T, 2).\text{tkval} = v' \)
   - \( (T, 1).\text{scr} = (T, 2).\text{scr} = e_{\text{TOK}} \)
   - \( (T, 1).\text{val} = (T, 2).\text{val} = 0 \)

   In both cases, in \( \Gamma_{S_0} \) there exist two deposits \( \{ A, v : \tau \} \) and \( \{ B, v' : \tau \} \) which did not exist in \( \Gamma_S \). The mapping \( txout \) inherits all the bindings of \( txout' \) except that for \( x \), and extends it with the bindings \( y \mapsto (T, 1) \) and \( y' \mapsto (T, 2) \). The mapping \( tkid \) is equal to \( tkid' \). The mapping \( txburn \) is equal to \( txburn' \).

3) \( \alpha = \text{join}(x, y, \mathcal{C}) \). In \( \Gamma_S \) there exist two deposits \( \{ A, v : \tau \} \) and \( \{ B, v' : \tau \} \), and \( \gamma = T \). The following conditions hold:
   - \( T \) has exactly two inputs
\[ v > 0 \]

\[ \langle A, 0 : \mathbb{B} \rangle_x | \Gamma \vdash_{x > gen(x,v)} \langle A, 0 : \mathbb{B} \rangle_x | A \vdash_{x > gen(x,v)} \Gamma \]

\[ \langle x = x_1 \cdots x_n \mid j \in 1..n \mid y \fresh \text{(except in burn auth for } x) \mid n = 1 \lor (n \geq 1 \land \forall i : \tau_i = \mathbb{B}) \]

\[ \left( \left\langle \xi \in 1..n \langle A_i, v_i : \tau_i \rangle_{x_i} \right| \Gamma \vdash_{x > split(x,v,y)} \left( \left\langle \xi \in 1..n \langle A_i, v_i : \tau_i \rangle_{x_i} \right| A_j \vdash_{x > burn(x,y)} \Gamma \right) \right) \]

\[ v, v' \geq 0 \]

\[ \langle A, (v + v') : \tau \rangle_x | \Gamma \vdash_{x > split(x,v,B)} \langle A, (v + v') : \tau \rangle_x | A \vdash_{x > split(x,v,B)} \Gamma \]

\[ \langle P, z \rangle \in \{ \langle A, x \rangle, (B, v) \} \]

\[ \langle A, v : \tau \rangle_x | \langle B, v' : \tau \rangle_y | \Gamma \vdash_{x > join(x,y,C)} \langle A, v : \tau \rangle_x | \langle B, v' : \tau \rangle_y | P \vdash_{x > join(x,y,C)} \Gamma \]

\[ \langle P, z \rangle \in \{ \langle A, x \rangle, (B, v) \} \mid \tau \neq \mathbb{B} \]

\[ \langle A, v : \tau \rangle_x | \langle B, v' : \tau \rangle_y | \Gamma \vdash_{x > xchg(x,y)} \langle A, v : \tau \rangle_x | \langle B, v' : \tau \rangle_y | P \vdash_{x > xchg(x,y)} \Gamma \]

\[ \langle A, v : \tau \rangle_x | \Gamma \vdash_{x > give(x,B)} \langle A, v : \tau \rangle_x | A \vdash_{x > give(x,B)} \Gamma \]

Fig. 8: Semantics of authorizations.

- \( \text{T}.in(1) = \txout'(x) \)
- \( \text{T}.in(2) = \txout'(y) \)
- \( \text{T} \) has exactly one output
- \( (T,1).owner = C \)

Further, if \( \tau = \mathbb{B} \), the following conditions hold:
- \( (T,1).val = v + v' \)
- \( (T,1).scr = \text{BTC} \)

Further, if \( \tau \neq \mathbb{B} \), the following conditions hold:
- \( (T,1).op = 3 \)
- \( (T,1).tkid = \txout'(x).tkid = \txout'(y).tkid \)
- \( (T,1).tkval = v + v' \)
- \( (T,1).scr = \text{BTC} \)
- \( (T,1).val = 0 \)

In both cases, in \( \Gamma_{S_0} \) there exist one deposit \( \langle A, (v + v') : \tau \rangle_x \) which did not exist in \( \Gamma_S \). The mapping \( \txout \) inherits all the bindings of \( \txout' \) except those for \( x \) and \( y \), and extends it with the binding \( z \mapsto (T,1) \). The mapping \( \txid \) equals to \( \txid' \). The mapping \( \txburn \) is equal to \( \txburn' \).

(4) \( \alpha = \xchg(x,y) \). In \( \Gamma_S \) there exist two deposits \( \langle A, v : \tau \rangle_x \) and \( \langle B, v' : \tau' \rangle_y, \tau \neq \mathbb{B}, \) and \( \gamma = T \). The following conditions hold:
- \( T \) has exactly two inputs and two outputs
- \( T.in(1) = \txout'(x) \)
- \( T.in(2) = \txout'(y) \)
- \( (T,1).op = 4 \)
- \( (T,1).owner = B \)
- \( (T,2).owner = A \)
- \( (T,1).tkid = \txout'(x).tkid \)
- \( (T,1).tkval = v \)
- \( (T,1).val = 0 \)
- \( (T,1).scr = \text{BTC} \)

Further, if \( \tau' = \mathbb{B} \), the following conditions hold:
- \( (T,2).val = v' \)
- \( (T,2).scr = \text{BTC} \)

Further, if \( \tau' \neq \mathbb{B} \), the following conditions hold:
- \( (T,1).op = 5 \)
- \( (T,1).tkid = \txout'(x).tkid \)
- \( (T,1).tkval = v \)
- \( (T,1).scr = \text{BTC} \)
- \( (T,1).val = 0 \)

In both cases, in \( \Gamma_{S_0} \) there exist a deposit \( \langle B, v : \tau \rangle_y \) which did not exist in \( \Gamma_S \).

b) The following conditions hold:
- \( T \) has exactly two inputs and two outputs
- \( (T,1).in = \txout'(x) \)
- \( (T,2).in \notin \text{ran}(\txout') \)
- \( (T,1).op = 4 \)
Fig. 9: Full token script.

```python
if not verrec(ptxo(1)) then  
    // ctzo is a token generator
    ctzo.op = 0  
    // gen action
    and ctzo.tkid = txid(ptxo(1))  
    // token id
    and ptxo(1).val = 0  
    // spent txo has 0 BTC
    and outlen(ctxo) = 1  
    // gen has 1 out
    and ctzo.tkval > 0  
    // positive token value
else true
    and

if rtzo.op = 1 then  
    //*************************************************************************  
    // BURN *************************************************************************  
    versig(ctzo.owner, rtzo.wit)  
    // check owner
    and verrec(rtzo(1))  
    // covenants on rtzo
    and inlen(rtzo(1)) = 1  
    // rtzo has 1 in
    and outlen(rtzo(1)) = 1  
    // rtzo has 1 out
    and rtzo(1).tkid = ctzo.tkid  
    // preserve token id
    and rtzo(1).tkval = ctzo.tkval  
    // preserve value
else if rtzo.op = 2 then  
    //*************************************************************************  
    // SPLIT *************************************************************************  
    versig(ctzo.owner, rtzo.wit)  
    // check owner
    and verrec(rtzo(1))  
    // covenants on rtzo
    and inlen(rtzo(1)) = 1  
    // rtzo has 1 in
    and outlen(rtzo(1)) = 1  
    // rtzo has 1 out
    and rtzo(1).tkid = ctzo.tkid  
    // preserve token id
    and rtzo(1).tkval = ctzo.tkval  
    // preserve value
else if rtzo.op = 3 then  
    //*************************************************************************  
    // JOIN  *************************************************************************  
    inlen(rtzo(1)) = 2  
    // rtzo has 2 ins
    and outlen(rtzo(1)) = 1  
    // rtzo has 1 out
    and verrec(rtzo(1))  
    // covenants on input 1
    and verrec(stxo(1))  
    // covenants on output 1
    and stxo.tkid = rtzo(1).tkid  
    // preserve token id
    and stxo.tkval = rtzo(1).tkval + stxo(2).tkval  
    // preserve value
else if rtzo.op = 4 then  
    //*************************************************************************  
    // Xchg  *************************************************************************  
    inlen(rtzo(1)) = 2  
    // rtzo has 2 ins
    and outlen(rtzo(1)) = 2  
    // rtzo has 2 outs
    and verrec(stxo.owner, rtzo.wit)  
    // check owner
    and verrec(stxo(1))  
    // covenants on input 1
    and verrec(stzo(1))  
    // covenants on output 1
    and stzo.tkid = stxo(1).tkid  
    // preserve token id
    and stzo.tkval = stxo(1).tkval  
    // preserve value
    if verrec(stzo(2)) then  
        //***** EXCHANGE TOKEN/TOKEN *****  
        versig(stzo.owner, rtzo.wit)  
        // check owner
        and rtzo(2).tkval = stzo(2).tkval  
        // preserve tkval
        and rtzo(2).tkid = stzo(2).tkid  
        // preserve tkid
    else  
        //***** EXCHANGE TOKEN/BTC *****  
        versig(stzo.owner, rtzo.wit), rtzo(2)
        and rtzo(2).val = stzo(2).val  
        // preserve BTC
else if rtzo.op = 5 then  
    //*************************************************************************  
    // Give  *************************************************************************  
    inlen(rtzo(1)) = 1  
    // rtzo has 1 in
    and outlen(rtzo(1)) = 1  
    // rtzo has 1 out
    and verrec(stzo.owner, rtzo.wit)  
    // check owner
    and rtzo(1).tkid = stzo.tkid  
    // preserve tkid
    and rtzo(1).tkval = stzo.tkval  
    // preserve value
else false
```
Note: in this case we are using the symbolic 

give

action to match the appending of a transaction \( T \) which consumes two inputs: the first one is mapped by \( txout' \)

and encodes a user-defined token \( \tau \), while the second one is not mapped by \( txout' \). So, the computational

action encoded by \( T \) is an \( xchg \) between \( \tau \) and some \( \tau' \). Appending \( T \) actually preserves the token identifier and its value, just changing its owner.

c) The following conditions hold:

- \( T \) has exactly two inputs and two outputs
- \((T, 1).\text{in} \notin \text{ran} (txout')\)
- \((T, 2).\text{in} = txout'(x)\)
- \((T, 1).\text{op} = 4\)
- \((T, 2).\text{SCR} = e \text{TOK}\)
- \((T, 2).\text{tkid} = txout'(x).\text{tkid}\)
- \((T, 2).\text{owner} = T.\text{in}(2).\text{owner}\)
- \((T, 2).\text{tkval} = v\)
- \((T, 2).\text{val} = 0\)

In all the subcases above, the mapping \( txout \) inherits all the bindings of \( txout' \) except that for \( x \), and extends it with the binding \( y \mapsto (T, 1) \) (in subcases (5a)-(5b)), or with \( y \mapsto (T, 2) \) (in subcase (5c)). The mapping \( tkid \) is equal to \( tkid' \). The mapping \( txburn \) is equal to \( txburn' \).

\( \alpha = \text{burn}(x, y) \). Let \( x = x_1 \ldots x_n \). In \( \Gamma_S \) there exist \( n \) deposits \( \langle A_i, v_i : \tau_i \rangle_{x_i} \), and \( \gamma = T = txburn'(y) \).

One of the following subcases must apply:

a) \( \tau_1 \neq 0 \) and \( n = 1 \). Further:

- \( T.\text{in} = txout'(x_1) \)
- \( T \) has exactly one output
- \((T, 1).\text{op} = 1\)
- \((T, 1).\text{SCR} = \text{false}\)
- \((T, 1).\text{val} = 0\)

b) \( \tau_1 = 0 \) and \( \tau_i = 0 \) for all \( i \in 1..n \). Further:

- \( T.\text{in} \) contains at least \( txout'(x_i) \) for all \( i \in 1..n \), but it does not contain any other output in \( \text{ran} (txout') \):

\( \{ x \mid txout'(x) \in T.\text{in} \} = \{ x_1, \ldots, x_n \} \)

- \( T \) does not correspond to any action \( \alpha \) discussed in the items (1)-(5) above.

The mapping \( txout \) is equal to \( txout' \). The mapping \( tkid \) is equal to \( tkid' \). The mapping \( txburn \) inherits all the bindings in \( txburn' \), except for that by \( y \).

\( \alpha = A \triangleright_x \text{gen}(x, v) \). In \( \Gamma_S \) there exists a deposit \( \langle A, 0 : 0 \rangle_{x} \), \( \gamma = B \mapsto * : m \), where \( m \) is the signature of \( A \) on a transaction \( T \) satisfying the conditions in item (1). The mappings \( txout, tkid, txburn \) are equal to \( txout', tkid', txburn' \), respectively.

\( \alpha = A \triangleright_x \text{split}(x, v, B) \). In \( \Gamma_S \) there exists a deposit \( \langle A, (v + v') : \tau \rangle_{x} \), \( \gamma = B \mapsto * : m \), where \( m \) is the signature of \( A \) on a transaction \( T \) satisfying the conditions in item (2). The mappings \( txout, tkid, txburn \) are equal to \( txout', tkid', txburn' \), respectively.

\( \alpha = P \triangleright_z \text{join}(x, y, C) \), where \( \langle P, z \rangle \in \{ \{ A, x \}, \{ B, y \} \} \), for some \( A \) and \( B \). In \( \Gamma_S \) there exist two deposits \( \langle A, v : \tau \rangle_{x} \) and \( \{ B, v' : \tau' \} \), with \( \tau \neq 0 \), \( \gamma = B \mapsto * : m \), where \( m \) is the signature of \( P \) on a transaction \( T \) satisfying the conditions in item (3). The mappings \( txout, tkid, txburn \) are equal to \( txout', tkid', txburn' \), respectively.

\( \alpha = A \triangleright_x \text{give}(x, B) \). In \( \Gamma_S \) there exists a deposit \( \langle A, v : \tau \rangle_{x} \), \( \gamma = B \mapsto * : m \), where \( m \) is the signature of \( A \) on a transaction \( T \) satisfying the conditions in item (5). The mappings \( txout, tkid, txburn \) are equal to \( txout', tkid', txburn' \), respectively.

\( \alpha = A_j \triangleright_x \text{burn}(x, y) \). Let \( x = x_1 \ldots x_n \). In \( \Gamma_S \) there exist \( n \) deposits \( \langle A_i, v_i : \tau_i \rangle_{x_i} \), and \( \gamma = T = txburn'(y) \).

If \( y \in \text{dom}(txburn') \), then \( txburn'(y) = T \), and \( txburn = txburn' \). Otherwise, \( txburn \) extends \( txburn' \) with the binding \( y \mapsto T \).

Inductive case 2: \( \text{coher}(\mathcal{S}, C \gamma, txout, tkid, txburn) \) holds if \( \text{coher}(\mathcal{S}, C, txout, tkid, txburn) \) and one of the following cases applies:

1. \( \gamma = T \), where no input of \( T \) belongs to \( \text{ran} txout \).
2. \( \gamma = A \mapsto * : m \), where \( \gamma \) does not correspond to any symbolic move, according to the inductive case 1.

APPENDIX C

SUPPLEMENTARY MATERIAL FOR SECTION VII

Proof of Lemma 3

By induction on \( S \sim C \). The base case is trivial. Otherwise, there are the following two cases:

- For inductive case 1 of the definition of coherence, we have \( S = S' \alpha \) and \( C = C' \gamma \), with \( S' \sim C' \). There are two subcases:
  - \( x \) occurs in \( S' \): by the induction hypothesis, \( txout'(x) \) is an output with the intended fields. We therefore only need to ensure that \( txout(x) = txout'(x) \). Inspecting how each \( \alpha \) is handled in the definition of coherence, we observe that this always holds except when \( x \) no
longer occurs in $\Gamma_S$. The latter case contradicts the hypothesis.

- $x$ does not occur in $S'$: Inspecting how each $\alpha$ is handled in the definition of coherence, we observe that this is only possible when $\gamma = T$ and $\text{txout}(x)$ points to some output of $T$ which has the intended fields.

- For inductive case 2 of the definition of coherence, we have $S = S'$ and $C = C'\gamma$, with $S' \sim C'$. By the induction hypothesis, $\text{txout}'(x)$ is an output with the intended fields. We therefore only need to ensure that $\text{txout}(x) = \text{txout}'(x)$, which holds since in inductive case 2 $\text{txout}' = \text{txout}$.

**Proof of Lemma 4**

By induction on $S \sim C$. The base case is trivial, since it only involves bitcoins. Otherwise, there are the following two cases:

- For inductive case 1 of the definition of coherence, we have $S = S'\alpha$ and $C = C'\gamma$, with $S' \sim C'$. There are two subcases:
  - $(T, i)$ also occurs (unspent) in $C'$: We now prove the items of the lemma:
    * when $(T, i).\text{tkid}$ does not correspond to any outputs, we apply the induction hypothesis to prove it is unspendable.
    * When its $\text{tkid}$ corresponds to some mapped output but $(T, i) \notin \text{ran}(\text{txout})$ (case 1), we also have $(T, i) \notin \text{ran}(\text{txout}')$, since otherwise $(T, i)$ would have been spent by $\gamma$ (as can be verified inspecting all the possible cases in the definition of coherence). Hence the induction hypothesis proves it unspendable.

- $(T, i)$ does not occur in $C'$: We have $\gamma = T$. Inspecting how each $\alpha$ is handled in the definition of coherence, we observe that this case is only possible when $(T, i).\text{tkid}$ is some output in $\text{ran}(\text{txout})$, which belongs to the blockchain. Further, inspecting the same cases, we observe that $(T, i)$ must also belong to $\text{ran}(\text{txout})$, it is spendable, and that it corresponds to a suitable symbolic deposit in $\Gamma_S$.

- For inductive case 2 of the definition of coherence, we have $S = S'$ and $C = C'\gamma$, with $S' \sim C'$. We also have $\text{txout}' = \text{txout}$ in this case. There are two subcases:
  - $(T, i)$ also occurs (unspent) in $C'$: the thesis immediately follows from the induction hypothesis, which involves the same $S$ and $\text{txout}$.
  - $(T, i)$ does not occur in $C'$: by the definition of coherence, this is possible only when $\gamma = T$ and no input of $T$ belongs to $\text{txout}$. We now prove the items of the lemma:
    * when $(T, i).\text{tkid}$ does not correspond to any output, we want to prove $(T, i)$ is unspendable. Indeed, the first part of the token script requires either that $(T, i).\text{tkid}$ is the parent output (contradiction) or that it has the same script and $\text{tkid}$. In the latter case, by the induction hypothesis, the parent was unspendable – contradiction.
    * When $(T, i).\text{tkid}$ corresponds to some mapped output but $(T, i) \notin \text{ran}(\text{txout})$ (case 1), we also want to prove it is unspendable.

**Proof of Theorem 5 (Computational soundness)**

Assume that $C$ satisfies the hypotheses, but there does not exist any $S$ which is coherent to $C$ and conforming to the symbolic strategies. Consider the longest prefix $C'$ of $C$ such that there exists some $S'$ which is coherent (to $C'$) and conforming (to the computational strategies). We have that $C = C'\gamma C''$. We now show that either $C'\gamma$ has a corresponding symbolic run $S'S''$ which is coherent and conforming to the symbolic strategies (contradicting the maximality of $C'$), or the adversary succeeded in a signature forgery (which can happen only with negligible probability). We proceed by cases on $\gamma$:

**Case 1 — $\gamma$ is a broadcast message:** Let $\gamma = A \rightarrow * : m$.

Then, either one of the items (7) to (12) of the inductive case 1 applies, or the inductive case 2 applies (since, by construction, the inductive case 2 catches all the other cases). In any case, we find a coherent extension $S''$ (possibly empty). Further, $S'S''$ conforms to the symbolic strategies: (i) if the inductive case 2 was applied, then $S''$ is empty, and so conformance follows from the induction hypothesis; (ii) if the inductive case 1 was applied, then the broadcast message $m$ is a signature of some user $B$. If $B$ is dishonest, then $\text{Adv}$ can compute the signature; then, there exists a symbolic strategy of $\text{Adv}$ which performs such move, and the resulting run conforms to the users’ symbolic strategies. If $B$ is honest, either the signature was forged by $\text{Adv}$ (but only with a negligible probability), or produced by $B$’s computational strategy. In the latter case, since the computational strategy of $B$ is derived from the symbolic one, $B$’s symbolic strategy outputs an authorization...
of $B$, which is exactly the label contained in $S''$. Therefore, $S'S''$ conforms to the symbolic strategies.

**Case 2 — $\gamma$ is a transaction:** Let $\gamma = T$. We consider the following subcases, according to the inputs of $T$:

- if none of the inputs of $T$ belongs to $\text{ran} (txout)$, then we achieve coherence and conformance with $S''$ empty.
- otherwise, if all the inputs of $T$ in $\text{ran} (txout)$ correspond to symbolic $\delta$ deposits, there are two subcases:
  - $T$ is obtained by one of the items from (1), (2), (3) or (5) of the inductive case 1 of coherence. Note that, by requirement (ii) of the definition of computational strategies, then all the witnesses in $T$ must have been broadcast in a previous computational step: by coherence, the corresponding authorizations have been performed in the symbolic run. Then, we can choose for $S''$ the symbolic action corresponding to $T$, which makes $S''$ a coherent extension of $S'$. Conformance holds trivially, by choosing Adv’s symbolic strategy to perform the symbolic action corresponding to $T$.
  - $T$ is obtained by the item (6) of the inductive case 1 of coherence (by construction, this catches all the other cases). The proof proceeds as in the previous case.

- otherwise, there exists at least one input of $T$ in $\text{ran} (txout)$ corresponding to a deposit of a user-defined token, say $\langle A, v : \tau \rangle_x$. Let $(T', i) = txout(x)$: since this output stores a user-defined token, then it must be $(T', i).\text{scr} = e_{\text{TOK}}$. Since $T$ spends $(T', i)$, then $e_{\text{TOK}}$ evaluates to true. We proceed by cases on $(T, 1).\text{op}$:
  - $\text{op} = 1$. In this case appending $T$ corresponds to performing a $\text{burn}$ action. The script $e_{\text{TOK}}$ in $(T', i)$ ensures that $T$ has only one input, i.e. $(T', i)$, and one outputs. Since $e_{\text{TOK}}$ requires the signature of $(T', i).\text{owner}$, and since by requirement (ii) of the definition of computational strategies, all the witnesses in $T$ must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorization (with a fresh name $y$) has been performed in the symbolic run. Let $S'' = \text{burn}(x, y)$. Then $S''$ is a valid extension of $S'$, since the preconditions of rule $[\text{burn}]$ are respected. Further, $S'S''$ is coherent with $C'\gamma$ by the item (6) of the inductive case 1 of coherence. Conformance holds trivially, by choosing Adv’s symbolic strategy to perform the symbolic action corresponding to $T$ (this also applies to all subcases below).
  - $\text{op} = 2$. In this case appending $T$ corresponds to performing a $\text{split}$ action. The script $e_{\text{TOK}}$ in $(T', i)$ ensures that:
    1) $T$ has only one input, i.e. $(T', i)$, and two outputs. Let $v_1 = (T, 1).\text{tkval}$, let $v_2 = (T, 2).\text{tkval}$, and let $B = (T, 2).\text{owner}$;
    2) $(T, 1).\text{scr} = e_{\text{TOK}}$;
    3) $v_1, v_2 \geq 0$, and $v_1 + v_2 = v$.

Since $e_{\text{TOK}}$ requires the signature of $(T', i).\text{owner} = A$, and since by requirement (ii) of the definition of computational strategies, all the witnesses in $T$ must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorization has been performed in the symbolic run. Let $S'' = \text{split}(x, v_1, B)$. Then $S''$ is a valid extension of $S'$, since the preconditions of rule $[\text{split}]$ are respected. Further, $S'S''$ is coherent with $C'\gamma$ by the item (2) of the inductive case 1 of coherence.

- $\text{op} = 3$. The script $e_{\text{TOK}}$ in $(T', i)$ ensures that:
  1) $T$ has two inputs and one output: let $(T'', j)$ be the other input, sibling of $(T', i)$;
  2) $(T, 1).\text{scr} = (T'', j).\text{scr} = e_{\text{TOK}}$;
  3) $(T', i).\text{tkid} = (T, 1).\text{tkid} = (T'', j).\text{tkid}$.

There are two further subcases, according to whether $(T'', j) \in \text{ran}(txout)$ or not:

* if $(T'', j) \in \text{ran}(txout)$, then this input corresponds to a deposit $\langle B, v' : \tau \rangle_y$, and appending $T$ corresponds to performing a $\text{join}$ action. By item 3 above, both $x$ and $y$ store the same token $\tau$. Since the script in $(T', i)$ requires the signature of $A$, and that in $(T'', j)$ requires the signature of $B$, and since by requirement (ii) of the definition of computational strategies, all the witnesses in $T$ must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorizations have been performed in the symbolic run. Let $C = (T, 1).\text{owner}$. Let $S'' = \text{join}(x, y, C)$. Then $S''$ is a valid extension of $S'$, since the preconditions of rule $[\text{join}]$ are respected. Further, $S'S''$ is coherent with $C'\gamma$ by the item (3) of the inductive case 1 of coherence.

* if $(T'', j) \notin \text{ran}(txout)$, then by the first item of Lemma 4, the output $(T'', j)$ is unspendable — contradiction. Then, this case does not apply.

- $\text{op} = 4$. The script $e_{\text{TOK}}$ in $(T', i)$ ensures that:
  1) $T$ has two inputs and two outputs: let $(T'', j)$ be the other input, sibling of $(T', i)$;
  2) $(T, 1).\text{scr} = e_{\text{TOK}}$.

There are two further subcases:

* if $(T'', j) \notin \text{ran}(txout)$, then appending $T$ corresponds to performing a $\text{give}$ action. Since the script in $(T', i)$ requires the signature of $A$, and since by requirement (ii) of the definition of computational strategies, all the witnesses in $T$ must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorization has been performed in the symbolic run. Let $B = (T'', j).\text{owner}$, and let $S'' = \text{give}(x, B)$. Then $S''$ is a valid extension of $S'$, since the preconditions of rule $[\text{give}]$ are respected. There are two subcases. If $(T'', j) = T.\text{in}(1)$, then $S'S''$ is coherent with $C'\gamma$ by the item (5)c of the inductive case 1 of coherence. Otherwise, if
\((T'', j) = \text{T.in}(2)\) we obtain coherence by the item (5)b of the inductive case 1 of coherence.

* otherwise, if \((T'', j) \in \text{ran}(txout)\), then this input corresponds to a deposit \((B, v' : \tau')_y\). Appending \(T\) corresponds to performing an \text{xchg} action. Since the script in \((T', i)\) requires the signature of \(A\), and that in \((T'', j)\) requires the signature of \(B\), and since by requirement (ii) of the definition of computational strategies, all the witnesses in \(T\) must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorizations have been performed in the symbolic run. There are two further subcases:

\[
\begin{align*}
&\text{· } \tau' = \emptyset. \text{Then, it must be } (T', i) = \text{T.in}(1) \text{ and } (T'', j) = \text{T.in}(2). \text{ Let } S'' = \text{xchg}(x, y). \\
&\text{· } \tau' \neq \emptyset. \text{If } (T', i) = \text{T.in}(1) \text{ and } (T'', j) = \text{T.in}(2) \text{ then let } S'' = \text{xchg}(x, y), \text{ otherwise let } S'' = \text{xchg}(y, x).
\end{align*}
\]

Then, \(S'S''\) is coherent with \(C'\gamma\) by the item (4) of the inductive case 1 of coherence.

– \(\text{op} = 5\). In this case appending \(T\) corresponds to performing a \text{give} action. The script \(e_{\text{TOK}}\) in \((T', i)\) ensures that:

1) \(T\) has only one input, i.e. \((T', i)\), and one output;
2) \((T, 1).\text{scr} = e_{\text{TOK}}\);
3) \((T, 1).\text{tkval} = (T', i).\text{tkval};
4) \((T, 1).\text{tkid} = (T', i).\text{tkid}.

Since \(e_{\text{TOK}}\) requires the signature of \((T', i).\text{owner} = A\), and since by requirement (ii) of the definition of computational strategies, all the witnesses in \(T\) must have been broadcast in a previous computational step, by coherence, it follows that the corresponding authorization has been performed in the symbolic run. Let \(S'' = \text{give}(x, B)\). Then \(S''\) is a valid extension of \(S'\), since the preconditions of rule [GIVE] are respected. Further, \(S'S''\) is coherent with \(C'\gamma\) by the item (5)a of the inductive case 1 of coherence.