Computationally sound Bitcoin tokens

Massimo Bartoletti
Università degli Studi di Cagliari
Cagliari, Italy
bart@unica.it

Stefano Lande
Università degli Studi di Cagliari
Cagliari, Italy
lande@unica.it

Roberto Zunino
Università degli Studi di Trento
Trento, Italy
roberto.zunino@unitn.it

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00022

***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 guarantee 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).

Due to space constraints, we provide the proofs of our statements in a separate technical report [13].

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:

$$\langle \mathsf{A}, v : \tau \rangle_x \qquad (v \in \mathbb{N})$$

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

$$\langle \mathsf{A}, 1 : \tau \rangle_x \mid \langle \mathsf{A}, 2 : \tau \rangle_y \mid \langle \mathsf{B}, 3 : ₿ \rangle_z$$

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 ₿. Performing this action (say, with $v = 10$) is modelled as a state transition:

$$\langle \mathsf{A}, 0 : ₿ \rangle_{x_0} \xrightarrow{gen} \langle \mathsf{A}, 10 : \tau \rangle_{x_1} \tag{1}$$

The label *gen* over the arrow records that the performed action is a token minting. The state transition (1) ensures that the identifier x_1 of the new deposit and the identifier τ of the minted token are *fresh*. After performing the action, A owns a deposit of ten units of the token τ. 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.:

$$\langle \mathsf{A}, 10 : \tau \rangle_{x_1} \xrightarrow{split} \langle \mathsf{A}, 8 : \tau \rangle_{x_2} \mid \langle \mathsf{B}, 2 : \tau \rangle_{x_3} \tag{2}$$

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:

$$\langle \mathsf{A}, 8 : \tau \rangle_{x_2} \xrightarrow{give} \langle \mathsf{B}, 8 : \tau \rangle_{x_4} \tag{3}$$

After that, B owns a total of 10 units of τ 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 τ. Now, B can *join* his two deposits, obtaining a single deposit with 10 units of τ. When performing the *join* action, B can also choose the owner of the new deposit, in this case transferring it back to A:

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{B}, 2 : \tau \rangle_{x_3} \xrightarrow{join} \langle \mathsf{A}, 10 : \tau \rangle_{x_5} \tag{4}$$

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

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{A}, 2 : \tau' \rangle_{x_6} \not\xrightarrow{join}$$

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

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{A}, 2 : \tau' \rangle_{x_6} \xrightarrow{xchg} \langle \mathsf{A}, 8 : \tau \rangle_{x_7} \mid \langle \mathsf{B}, 2 : \tau' \rangle_{x_8}$$

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 τ' from B for 1₿:

$$\langle \mathsf{B}, 2 : \tau' \rangle_{x_8} \mid \langle \mathsf{A}, 1 : ₿ \rangle_{x_9} \xrightarrow{xchg} \langle \mathsf{A}, 2 : \tau' \rangle_{x_{10}} \mid \langle \mathsf{B}, 1 : ₿ \rangle_{x_{11}}$$

Finally, a user can *burn* any of her deposits. This is rendered as a state transition where the burnt deposits are no longer present in the target configuration. For instance, starting from the configuration above, A can burn her 2 units of τ' in x_{10}:

$$\langle \mathsf{A}, 2 : \tau' \rangle_{x_{10}} \mid \langle \mathsf{B}, 1 : ₿ \rangle_{x_{11}} \xrightarrow{burn} \langle \mathsf{B}, 1 : ₿ \rangle_{x_{11}}$$

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:

T_1
in(1): $\bot$
wit(1): $\bot$
out(1): {scr : versig(pk_{A}, rtx.wit), val : 10₿}

In general, the 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 out field is a record, where scr is a *script*, and 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 versig(pk_{A}, rtx.wit) verifies a signature on the redeeming transaction (rtx, excluding its wit field) against A's public key pk_{A}. This signature is retrieved from the wit field of 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 $\langle \mathsf{A}, 10 : ₿ \rangle_{x_1}$.

To perform the *split* action (2) on $\tau = ₿$, we can spend T_1 with a transaction T_2 with *two* outputs:

T_2
in(1): (T_1, 1)
wit(1): $sig_{sk_{\mathsf{A}}}(\mathsf{T}_1)$
out(1): {scr : versig(pk_{A}, rtx.wit), val : 8₿}
out(2): {scr : versig(pk_{B}, rtx.wit), val : 2₿}

The first output, that we denote by $(\mathsf{T}_2, 1)$, corresponds to the deposit $\langle \mathsf{A}, 8 : \ddot{\mathbb{B}} \rangle_{x_2}$ in (2). Instead, the output $(\mathsf{T}_2, 2)$ corresponds to $\langle \mathsf{B}, 2 : \ddot{\mathbb{B}} \rangle_{x_3}$. These outputs can be spent independently. For instance, performing the *give* action in (3) corresponds to appending a transaction which spends $(\mathsf{T}_2, 1)$:

T_3
$\mathsf{in}(1)$: $(\mathsf{T}_2, 1)$
$\mathsf{wit}(1)$: $sig_{sk_\mathsf{A}}(\mathsf{T}_3)$
$\mathsf{out}(1)$: $\{\mathsf{scr} : \mathsf{versig}(pk_\mathsf{B}, \mathsf{rtx.wit}), \mathsf{val} : 8\ddot{\mathbb{B}}\}$

At this point, we have two unspent outputs on the blockchain: $(\mathsf{T}_2, 2)$ and $(\mathsf{T}_3, 1)$. We can perform the *join* action in (4) by spending both of them *simultaneously* with the following transaction, which has *two* inputs:

T_4	
$\mathsf{in}(1)$: $(\mathsf{T}_2, 2)$	$\mathsf{in}(2)$: $(\mathsf{T}_3, 1)$
$\mathsf{wit}(1)$: $sig_{sk_\mathsf{B}}(\mathsf{T}_4)$	$\mathsf{wit}(2)$: $sig_{sk_\mathsf{B}}(\mathsf{T}_4)$
out: $\{\mathsf{scr} : \mathsf{versig}(pk_\mathsf{B}, \mathsf{rtx.wit}), \mathsf{val} : 10\ddot{\mathbb{B}}\}$	

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 [14]. The technique relies on *covenants*, an extension of Bitcoin scripts which allows transactions to constrain the scripts of the redeeming ones. For instance, let e be an arbitrary script. A transaction output containing the script:

$$e \text{ and } \mathsf{verrec}(\mathsf{rtxo}(n))$$

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 $\mathsf{verrec}(\mathsf{rtxo}(n))$.

Using covenants, we can mint a token by appending the transaction T below, where the extra field arg is syntactic sugar for a *sequence* of values accessible by the script (we denote with $\mathsf{arg}.i$ the i-th element of this sequence):

T
...
$\mathsf{out}(1)$: $\{\mathsf{arg} : pk_\mathsf{A}$,
$\quad\mathsf{scr} : \mathsf{versig}(\mathsf{ctxo.arg}.1, \mathsf{rtx.wit})$ and // verify signature
$\quad\quad\mathsf{rtxo}(1).\mathsf{val} = 1$ and // preserve value
$\quad\quad\mathsf{verrec}(\mathsf{rtxo}(1))$, // preserve script
$\quad\mathsf{val} : 1\ddot{\mathbb{B}}\}$

The 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 arg to B's public key. For this to be possible, T' must satisfy the conditions specified in T's script: (i) the wit field must contain the signature of the current owner; (ii) the output at index 1 must have $1\ddot{\mathbb{B}}$ 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 arg sequence, to represent the number of token units deposited in the transaction output. We can implement a splittable token as follows:

T_{split}
...
$\mathsf{out}(1)$: $\{\mathsf{arg} : pk_\mathsf{A}\ v$,
$\quad\mathsf{scr} : \mathsf{versig}(\mathsf{ctxo.arg}.1, \mathsf{rtx.wit})$ and
$\quad\quad\mathsf{rtxo}(1).\mathsf{arg}.2 + \mathsf{rtxo}(2).\mathsf{arg}.2 = \mathsf{ctxo.arg}.2$ and
$\quad\quad\mathsf{verrec}(\mathsf{rtxo}(1))$ and $\mathsf{verrec}(\mathsf{rtxo}(2))$ and
$\quad\quad\mathsf{outlen}(\mathsf{rtx}) = 2$
$\quad\mathsf{val} : \cdots\}$

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

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

$$e \triangleq \text{ if } \mathsf{rtxo}(1).\mathsf{arg}.3 = S \text{ then } e_{split} \text{ else } e_{join}$$

where e_{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 $\langle \mathsf{B}, 8 : \tau \rangle_{x_4}$ and $\langle \mathsf{B}, 2 : \tau \rangle_{x_3}$. 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 *viceversa*. 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., $(\mathsf{T}_3, 1)$ is the sibling of $(\mathsf{T}_4, 1)$ when appending T_5). However, neither Bitcoin nor its extensions with covenants [12], [14]–[16] 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:

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

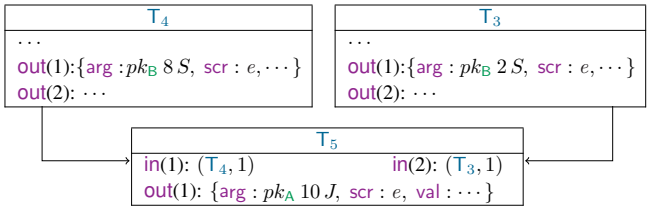

Fig. 1: A transaction T_5 attempting to join T_4 and T_3.

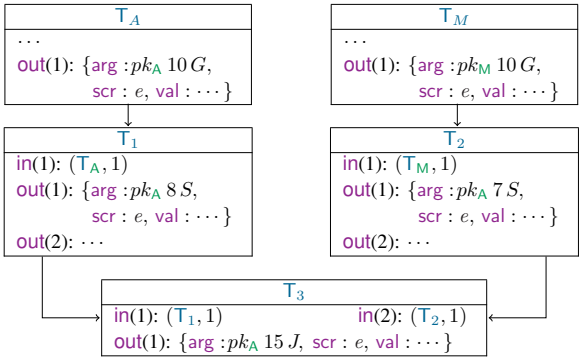

Fig. 2: A *join* attack merging two different tokens.

Using this new operator, we can encode the conditions (iii) and (iv) in e_{join} as follows:

$$\mathsf{rtxo}(1).\mathsf{arg}.2 = \mathsf{stxo}(1).\mathsf{arg}.2 + \mathsf{stxo}(2).\mathsf{arg}.2 \qquad \text{(iii)}$$

$$\mathsf{verrec}(\mathsf{stxo}(1)) \text{ and } \mathsf{verrec}(\mathsf{stxo}(2)) \qquad \text{(iv)}$$

Although this implementation of e_{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_{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 *join* attack of Figure 2, and eventually obtain a secure and efficient implementation of fungible tokens. We now sketch the script e_{TOK} which implements tokens. First, we add a fourth element to the arg sequence, to record in each transaction output the *identifier* of the token deposited in that output. As

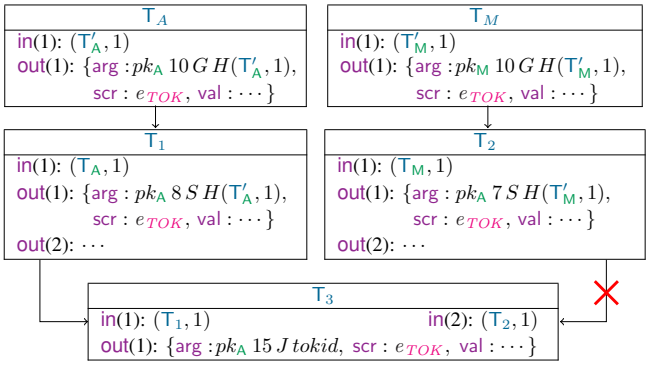

Fig. 3: Thwarting a *join* attack.

an identifier, we use the hash of the *parent* of the minting transaction, which we access through the script $\mathsf{txid}(\mathsf{ptxo}(1))$. When the script in a minting transaction (e.g., T_A and T_M in Figure 2) is evaluated, it ensures that the arg field of the minting transaction actually contains the token identifier:

$$e_{gen} \triangleq \cdots \text{ and } \mathsf{ctxo}.\mathsf{arg}.4 = \mathsf{txid}(\mathsf{ptxo}(1))$$

Then, the sub-scripts corresponding to all other token actions check that the redeeming transaction preserves the token identifier. For instance, in the *join* sub-script, besides checking conditions (iii) and (iv) as shown before, we add the condition:

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

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

Thwarting forgery attacks: Note that minting a token amounts to appending a *gen* transaction. For instance, in Figure 4, A appends the *gen* transaction T_A, which mints 10 units of a fresh token with identifier $H(\mathsf{T}'_A, 1)$. To validate this operation, the script e_{BTC} in T'_A is executed, verifying only that T_A is signed by A. Consequently, when minting a token, A can choose arbitrary values for the other fields of the transaction T_A: in particular, A could choose a token identifier (i.e., the value in arg.4) which is different from the correct one.

For instance, Figure 4 shows the adversary M *forging* 10 additional units of the token with identifier $H(\mathsf{T}'_A, 1)$. This is achieved by appending the transaction T_M, which spends a standard transaction T'_M, and setting its arg.4 field to $H(\mathsf{T}'_A, 1)$. However, these forged token units are *unspendable*: when M attempts to spend these units by appending T_2, the e_{TOK} script in T_M is finally executed. There, the check $\mathsf{ctxo}.\mathsf{arg}.4 = \mathsf{txid}(\mathsf{ptxo}(1))$ in e_{gen} fails, because $\mathsf{ctxo}.\mathsf{arg}.4$ evaluates to $H(\mathsf{T}'_A, 1)$, while $\mathsf{txid}(\mathsf{ptxo}(1))$ evaluates to $H(\mathsf{T}'_M, 1)$. In general, although forging tokens is unavoidable, the script e_{TOK} ensures that forged tokens are unspendable, so making forgery immaterial.

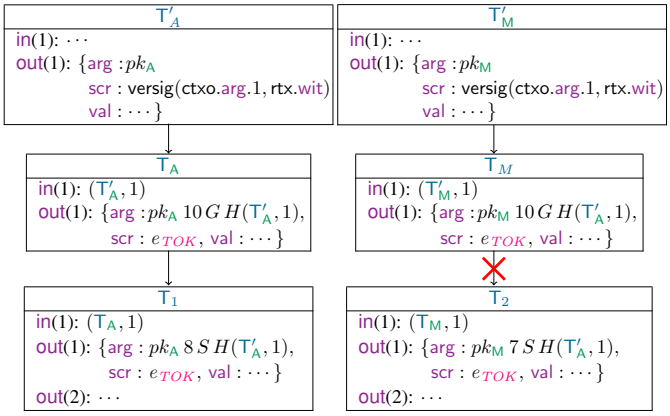

Fig. 4: Thwarting a forgery attack.

III. A SYMBOLIC MODEL OF TOKENS

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

- $gen(x, v)$ 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.
- $burn(\boldsymbol{x}, y)$ represents the act of destroying a sequence of deposits $\boldsymbol{x}$, moving them to an unspendable deposit y.
- $split(x, v, \mathsf{B})$ represents the act of splitting a deposit x (say, containing $v + v'$ units of a token τ) in two deposits of τ. 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.
- $join(x, y, \mathsf{C})$ represents the joining of two deposits x and y of the same token, into a new deposit, owned by C.
- $xchg(x, y)$ 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 ₿, this action represents buying/selling tokens for bitcoins.
- $give(x, \mathsf{B})$ 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* Γ is a compositions of deposits and authorizations. We assume that configurations form a commutative monoid under the composition operator $|$, and we use $\mathbf{0}$ to denote the empty configuration. We require that if $\langle \mathsf{A}, v : \tau \rangle_x$ and $\langle \mathsf{B}, v' : \tau' \rangle_{x'}$ both occur in Γ, then $x \neq x'$. We define a transition semantics between configurations in Figure 5. Transitions are decorated with labels, which describe the performed actions. The rules for granting authorizations (Figure 6 are straightforward.

Rule [GEN] consumes a bitcoin deposit x owned by A to generate v units of a new token τ, 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 [BURN] removes from the configuration a single token deposit (when $n = 1$ and $\tau_1 \neq$ ₿), or atomically removes a sequence of bitcoin deposits (when $\tau_i =$ ₿ for all i). Rule [SPLIT] divides a deposit x in two fresh deposits y and z, preserving the number of token units. Rule [JOIN] allows A and B to merge two deposits of a token τ, preserving the amount of token units, and transferring the new deposit to C. Rule [XCHG] allows A and B to swap two deposits, containing either user-defined tokens or bitcoins. Finally, rule [GIVE] allows A to donate one of her deposits to another user.

The transition relation $\to$ is non-deterministic, because of the fresh names generated for deposits and tokens: however, given a transition $\Gamma \xrightarrow{\alpha} \Gamma'$ the label α is uniquely determined from Γ and Γ'. A *symbolic run* $\mathcal{S}$ is a (possibly infinite) sequence $\Gamma_0 \Gamma_1 \cdots$, where Γ_0 contains only ₿ deposits, and for all $i \geq 0$ there exists some (unique) α_i such that $\Gamma_i \xrightarrow{\alpha_i} \Gamma_{i+1}$. For all $i \geq 0$, we denote with $\mathcal{S}_i$ the i-th element of the run, when this element exists. If $\mathcal{S}$ is finite, we denote its length as $|\mathcal{S}|$, and we write $\Gamma_{\mathcal{S}}$ for its last configuration, i.e. $\mathcal{S}_{|\mathcal{S}|-1}$.

Definition 1 (Token balance)**.** We define the balance of a token $\tau \neq$ ₿ in a configuration Γ inductively as follows:

$$bal_\tau(\mathbf{0}) = 0 \qquad bal_\tau(\Gamma \mid \Gamma') = bal_\tau(\Gamma) + bal_\tau(\Gamma')$$
$$bal_\tau(\langle \mathsf{A}, v : \tau \rangle_x) = v \qquad bal_\tau(\langle \mathsf{A}, v : \tau' \rangle_x) = 0 \quad (\tau' \neq \tau)$$

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:

$$minted_\tau(\mathcal{S}) = v \text{ if } \exists i : \begin{array}{l} \mathcal{S}_i \xrightarrow{gen(x,v)} \mathcal{S}_{i+1}, \text{ and} \\ \tau \text{ occurs in } \mathcal{S}_{i+1} \text{ but not in } \mathcal{S}_i \end{array}$$

$$burnt_\tau(\mathcal{S}) = \sum \left\{ v \;\middle|\; \exists i : \begin{array}{l} \mathcal{S}_i \xrightarrow{burn(x,y)} \mathcal{S}_{i+1}, \text{ and} \\ \mathcal{S}_i = \Gamma \mid \langle \mathsf{A}, v : \tau \rangle_x \end{array} \right\}$$

Lemma 1. *Let $\mathcal{S}$ be a finite symbolic run. For all $\tau \neq$ ₿:*

$$bal_\tau(\mathcal{S}) \;=\; minted_\tau(\mathcal{S}) - burnt_\tau(\mathcal{S})$$

Proof. By induction on $\mathcal{S}$, 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 ([GEN]) and burning ([BURN]), which are explicitly taken into account by the equation. □

IV. BITCOIN TRANSACTIONS

In this section we recall the functionality of Bitcoin. To this purpose we rely on the formal model of [17], 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).

$$\frac{\Gamma = \mathsf{A} \rhd_x gen(x,v) \mid \Gamma' \quad v > 0 \quad y, \tau \text{ fresh}}{\langle \mathsf{A}, 0 : \ddot{\mathbb{B}} \rangle_x \mid \Gamma \xrightarrow{gen(x,v)} \langle \mathsf{A}, v : \tau \rangle_y \mid \Gamma'} [\textsc{Gen}] \qquad \frac{\Gamma = \left(\parallel_{i \in 1..n} \mathsf{A}_i \rhd_{x_i} burn(x_1 \cdots x_n, y) \right) \mid \Gamma' \quad n = 1 \vee (n \geq 1 \wedge \forall i : \tau_i = \ddot{\mathbb{B}})}{\left(\parallel_{i \in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \mid \Gamma \xrightarrow{burn(x_1 \cdots x_n, y)} \Gamma'} [\textsc{Burn}]$$

$$\frac{\Gamma = \mathsf{A} \rhd_x split(x, v, \mathsf{B}) \mid \Gamma' \quad v, v' \geq 0 \quad y, y' \text{ fresh}}{\langle \mathsf{A}, (v + v') : \tau \rangle_x \mid \Gamma \xrightarrow{split(x,v,\mathsf{B})} \langle \mathsf{A}, v : \tau \rangle_y \mid \langle \mathsf{B}, v' : \tau \rangle_{y'} \mid \Gamma'} [\textsc{Split}] \qquad \frac{\Gamma = \mathsf{A} \rhd_x join(x, y, \mathsf{C}) \mid \mathsf{B} \rhd_y join(x, y, \mathsf{C}) \mid \Gamma' \quad z \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau \rangle_y \mid \Gamma \xrightarrow{join(x,y,\mathsf{C})} \langle \mathsf{C}, (v + v') : \tau \rangle_z \mid \Gamma'} [\textsc{Join}]$$

$$\frac{\Gamma = \mathsf{A} \rhd_x xchg(x, y) \mid \mathsf{B} \rhd_y xchg(x, y) \mid \Gamma' \quad \tau \neq \ddot{\mathbb{B}} \quad x', y' \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau' \rangle_y \mid \Gamma \xrightarrow{xchg(x,y)} \langle \mathsf{A}, v' : \tau' \rangle_{x'} \mid \langle \mathsf{B}, v : \tau \rangle_{y'} \mid \Gamma'} [\textsc{Xchg}] \qquad \frac{\Gamma = \mathsf{A} \rhd_x give(x, \mathsf{B}) \mid \Gamma' \quad y \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \Gamma \xrightarrow{give(x,\mathsf{B})} \langle \mathsf{B}, v : \tau \rangle_y \mid \Gamma'} [\textsc{Give}]$$

Fig. 5: Semantics of token actions.

$$\frac{v > 0}{\langle \mathsf{A}, 0 : \ddot{\mathbb{B}} \rangle_x \mid \Gamma \xrightarrow{\mathsf{A} \rhd_x gen(x,v)} \langle \mathsf{A}, 0 : \ddot{\mathbb{B}} \rangle_x \mid \mathsf{A} \rhd_x gen(x, v) \mid \Gamma} [\textsc{AuthGen}]$$

$$\frac{\boldsymbol{x} = x_1 \cdots x_n \quad j \in 1..n \quad y \text{ fresh (except in burn auth for } \boldsymbol{x}) \quad n = 1 \vee (n \geq 1 \wedge \forall i : \tau_i = \ddot{\mathbb{B}})}{\left(\parallel_{i \in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \mid \Gamma \xrightarrow{\mathsf{A}_j \rhd_{x_j} burn(\boldsymbol{x}, y)} \left(\parallel_{i \in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \mid \mathsf{A}_j \rhd_{x_j} burn(\boldsymbol{x}, y) \mid \Gamma} [\textsc{AuthBurn}]$$

$$\frac{v, v' \geq 0}{\langle \mathsf{A}, (v + v') : \tau \rangle_x \mid \Gamma \xrightarrow{\mathsf{A} \rhd_x split(x,v,\mathsf{B})} \langle \mathsf{A}, (v + v') : \tau \rangle_x \mid \mathsf{A} \rhd_x split(x, v, \mathsf{B}) \mid \Gamma} [\textsc{AuthSplit}]$$

$$\frac{(\mathsf{P}, z) \in \{(\mathsf{A}, x), (\mathsf{B}, y)\}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau \rangle_y \mid \Gamma \xrightarrow{\mathsf{P} \rhd_z join(x,y,\mathsf{C})} \langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau \rangle_y \mid \mathsf{P} \rhd_z join(x, y, \mathsf{C}) \mid \Gamma} [\textsc{AuthJoin}]$$

$$\frac{(\mathsf{P}, z) \in \{(\mathsf{A}, x), (\mathsf{B}, y)\} \quad \tau \neq \ddot{\mathbb{B}}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau' \rangle_y \mid \Gamma \xrightarrow{\mathsf{P} \rhd_z xchg(x,y)} \langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau' \rangle_y \mid \mathsf{P} \rhd_z xchg(x, y) \mid \Gamma} [\textsc{AuthExchange}]$$

$$\frac{}{\langle \mathsf{A}, v : \tau \rangle_x \mid \Gamma \xrightarrow{\mathsf{A} \rhd_x give(x,\mathsf{B})} \langle \mathsf{A}, v : \tau \rangle_x \mid \mathsf{A} \rhd_x give(x, \mathsf{B}) \mid \Gamma} [\textsc{AuthGive}]$$

Fig. 6: Semantics of authorizations.

Transactions: Following the formalization in [17], we represent transactions as records with the following fields:[1].

- in is the list of *inputs*. Each of these inputs is a *transaction output* (T, i), referring to the i-th output field of T.
- wit is the list of *witnesses*, of the same length as in. Intuitively, for each (T, i) in the in field, the witness at the same index must make the i-th output script of T evaluate to true.
- out is the list of *outputs*. Each output is a record of the form $\{\mathsf{scr} : e, \mathsf{val} : v\}$, where e is a script, and $v \geq 0$.

We let f range over transaction fields, and we denote with $\mathsf{T}.\mathsf{f}$ the content of field f of transaction T. We write $\mathsf{T}.\mathsf{f}(i)$ for the i-th element of the sequence $\mathsf{T}.\mathsf{f}$, when in range; when the sequence has exactly one element, we write $\mathsf{T}.\mathsf{f}$ for $\mathsf{T}.\mathsf{f}(1)$. For transaction outputs (T, i), we interchangeably use the notation (T, i) and $\mathsf{T}.\mathsf{out}(i)$, and we use the notation above to access their sub-fields scr and 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 $\mathsf{versig}(\mathsf{A}, e)$ for $\mathsf{versig}(pk_{\mathsf{A}}, e)$, and $sig_{\mathsf{A}}(\mathsf{T})$ for $sig_{sk_{\mathsf{A}}}(\mathsf{T})$.

[1]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.

Scripts: Bitcoin scripts are terms with the following syntax:

$e ::= v$	constant (integer or bitstring)
$\mid e \circ e$	operators $(\circ \in \{+, -, =, <\})$
$\mid$ if e then e else e	conditional
$\mid e.n$	n-th element of sequence e $(n \in \mathbb{N})$
$\mid$ rtx.wit	witnesses of the redeeming tx
$\mid \lvert e \rvert$	size (number of bytes)
$\mid \mathsf{H}(e)$	hash
$\mid \mathsf{versig}(e, e')$	signature verification

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 (rtx.wit); further, they can compute the size $|e|$ of a bitstring and its hash $\mathsf{H}(e)$. The script $\mathsf{versig}(e, e')$ 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.[2] For all signatures, the signed message is the redeeming transaction (except its witnesses).

The semantics of scripts is in Figure 7. The script evaluation function $[\![\cdot]\!]_{\mathsf{T},i}$ takes two additional parameters: T is the

[2]Multi-signature verification is supported by Bitcoin scripts, but immaterial for our technical development: therefore, we omit it.

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 $size(v)$ the size (in bytes) of an integer v, and with ver a signature verification function (the definition of these semantic operators is standard, see e.g. [17]). The semantics of a script can be undefined, e.g. when accessing an element of a non-sequence: we denote this case as $\bot$. All the operators are *strict* i.e. they evaluate to $\bot$ if some of their operands is $\bot$. We use standard syntactic sugar for scripts, e.g.: (i) $false \triangleq 0$, (ii) $true \triangleq 1$, (iii) e and $e' \triangleq$ if e then e' else $false$, (iv) e or $e' \triangleq$ if e then $true$ else e', and finally (v) not $e \triangleq$ if e then $false$ else $true$.

Blockchains: A blockchain $\mathbf{B}$ is a finite sequence $\mathsf{T}_0 \cdots \mathsf{T}_n$, where T_0 is the only coinbase transaction (i.e., $\mathsf{T}_0.\mathsf{in} = \bot$). We say that the transaction output (T_i, j) is *spent* in $\mathbf{B}$ iff there exists some $\mathsf{T}_{i'}$ in $\mathbf{B}$ (with $i' > i$) and some j' such that $\mathsf{T}_{i'}.\mathsf{in}(j') = (\mathsf{T}_i, j)$. The *unspent transaction outputs* of $\mathbf{B}$, written $UTXO(\mathbf{B})$, is the set of transaction outputs (T_i, j) which are unspent in $\mathbf{B}$. A transaction T is a *valid extension* of $\mathbf{B} = \mathsf{T}_0 \cdots \mathsf{T}_n$ whenever the following conditions hold:

1) for each input i of T, if $\mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j)$ then:
   - $\mathsf{T}' = \mathsf{T}_h$, for some $h < n$ (i.e., T' is in $\mathbf{B}$);
   - the output (T', j) is not spent in $\mathbf{B}$;
   - $[\![(\mathsf{T}', j).\mathsf{scr}]\!]_{\mathsf{T},i} = v \neq 0$;
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 $\mathsf{T}_0 \cdots \mathsf{T}_{i-1}$. The difference between the amount of inputs and that of outputs of transactions is the *fee* paid to miners who participate in the consensus protocol.

V. Neighbourhood Covenants

To extend Bitcoin with neighbourhood covenants, 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 $\{\mathsf{arg} : \boldsymbol{a}, \mathsf{scr} : e, \mathsf{val} : v\}$, where $\boldsymbol{a}$ 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 covenants.

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

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

More precisely, consider the case where a transaction output (T', j) is redeemed by a transaction T, through its i-th input. When used within the script of (T', j): $\mathsf{rtxo}(n)$ refers to the n-th output of T; $\mathsf{stxo}(n)$ refers to the output redeemed by the n-th input of T; $\mathsf{ptxo}(n)$ refers to the output redeemed by the n-th input of T'. In Figure 8, we exemplify these outputs in relation to the transaction T_c. The semantics of o is defined in the first line of Figure 9; its result is a pair (T, n).

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

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

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

$$\mathsf{ctxo} \triangleq \mathsf{stxo}(\mathsf{inidx})$$

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

Example 1. Consider the transactions in Figure 8. The script in $\mathsf{T}_c.\mathsf{out}(1)$ checks that (i) the arg field of ctxo, i.e. the same output which contains the script, equals to n_c; (ii) the arg field of $\mathsf{ptxo}(1)$, i.e. the parent transaction output redeemed by $\mathsf{T}_c.\mathsf{in}(1)$, equals to n_p; (iii) the arg field of $\mathsf{rtxo}(3)$, i.e. the third output of the redeeming transaction T_r, equals to n_r; (iv) the arg field of $\mathsf{stxo}(2)$, i.e. the sibling transaction output redeemed by $\mathsf{T}_r.\mathsf{in}(2)$, equals to n_s. Note that, in general, any script used in T_c 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. $\diamond$

Figure 9 defines the semantics of extended scripts. As in Section IV, the function $[\![\cdot]\!]$ takes as parameters the redeeming 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. Indeed, this is also true without exploiting the $\mathsf{stxo}()$ and $\mathsf{ptxo}()$ operators, i.e. by only using

$$[\![v]\!]_{\mathsf{T},i} = v \qquad [\![e \circ e']\!]_{\mathsf{T},i} = [\![e]\!]_{\mathsf{T},i} \circ_\bot [\![e']\!]_{\mathsf{T},i} \qquad [\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_{\mathsf{T},i} = \mathit{if}\ [\![e_0]\!]_{\mathsf{T},i}\ \mathit{then}\ [\![e_1]\!]_{\mathsf{T},i}\ \mathit{else}\ [\![e_2]\!]_{\mathsf{T},i}$$

$$[\![e.n]\!]_{\mathsf{T},i} = v_n \ \text{ if } [\![e]\!]_{\mathsf{T},i} = v_1 \cdots v_k \ (1 \le n \le k) \qquad [\![\mathsf{rtx.wit}]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{wit}(i)$$

$$[\![|e|]\!]_{\mathsf{T},i} = \mathit{size}([\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{H}(e)]\!]_{\mathsf{T},i} = H([\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{versig}(e, e')]\!]_{\mathsf{T},i} = \mathsf{ver}_{[\![e]\!]_{\mathsf{T},i}}([\![e']\!]_{\mathsf{T},i}, \mathsf{T}, i)$$

Fig. 7: Semantics of Bitcoin scripts.

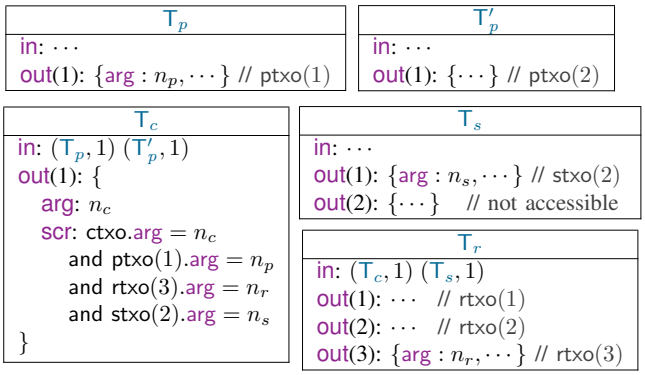

Fig. 8: Accessing transaction outputs through a script.

covenants between the current and the redeeming transaction. To prove this, we describe how to simulate in the extended Bitcoin any counter machine [18], 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: inc i increments register i, dec i decrements it, zero i sets it to zero, if $i \neq 0$ goto j conditionally jumps to instruction j when register i is not zero, 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 ₿ 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. (*sketch*) 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 arg field of the transaction output o, storing it as a sequence of integers. As a shorthand, we write $\mathsf{o.r}_i$ for $\mathsf{o.arg}.i$, and o.p for $\mathsf{o.arg}.(n+1)$. 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 at position ctxo.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 ctxo.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 verrec on the redeeming transaction to ensure the last part. Then, we check that the new state in $\mathsf{T}.\mathsf{out}(1).\mathsf{arg}$ agrees with the counter machine semantics, by cases on the instruction pointed to by o.p. If the instruction is inc i, then we require $\mathsf{rtxo}(1).\mathsf{r}_i = \mathsf{ctxo}.\mathsf{r}_i + 1$, $\mathsf{rtxo}(1).\mathsf{r}_k = \mathsf{ctxo}.\mathsf{r}_k$ for all $k \neq i$, and $\mathsf{rtxo}(1).\mathsf{p} = \mathsf{ctxo}.\mathsf{p} + 1$. The dec i and zero i cases are analogous. For the instruction if $i \neq 0$ goto j, we check the value of $\mathsf{ctxo}.\mathsf{r}_i$: if nonzero, we require that $\mathsf{rtxo}(1).\mathsf{p} = j$, otherwise that $\mathsf{rtxo}(1).\mathsf{p} = \mathsf{ctxo}.\mathsf{p}+1$. In both cases, we require that $\mathsf{rtxo}(1).\mathsf{r}_k = \mathsf{ctxo}.\mathsf{r}_k$ 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 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* [19]–[21]. □

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

$$[\![\mathsf{rtxo}(e)]\!]_{\mathsf{T},i} = (\mathsf{T}, [\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{stxo}(e)]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{in}([\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{ptxo}(e)]\!]_{\mathsf{T},i} = \mathsf{T}'.\mathsf{in}([\![e]\!]_{\mathsf{T},i}) \text{ if } \mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j)$$

$$[\![\mathsf{o}.\mathsf{f}]\!]_{\mathsf{T},i} = [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{f} \qquad [\![\mathsf{verscr}(\mathsf{o}, e)]\!]_{\mathsf{T},i} = e \equiv [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{scr} \qquad [\![\mathsf{verrec}(\mathsf{o})]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{in}(i).\mathsf{scr} \equiv [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{scr} \qquad [\![\mathsf{txid}(\mathsf{o})]\!]_{\mathsf{T},i} = H([\![\mathsf{o}]\!]_{\mathsf{T},i})$$

$$[\![\mathsf{outidx}]\!]_{\mathsf{T},i} = j \text{ if } \mathsf{T}.\mathsf{in}(i) = (\mathsf{T}, j) \qquad [\![\mathsf{inidx}]\!]_{\mathsf{T},i} = i \qquad [\![\mathsf{outlen}(\mathsf{o})]\!]_{\mathsf{T},i} = |\mathsf{T}'.\mathsf{out}| \qquad [\![\mathsf{inlen}(\mathsf{o})]\!]_{\mathsf{T},i} = |\mathsf{T}'.\mathsf{in}| \text{ if } [\![\mathsf{o}]\!]_{\mathsf{T},i} = (\mathsf{T}', j)$$

Fig. 9: Semantics of neighbourhood covenants (extending Figure 7).

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 [14], 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 [22]. 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* $\mathcal{C}$ is a sequence of bitstrings γ, each of which encodes one of the following actions: (i) $\mathsf{A} \to * : m$, 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 T_0. By extracting the transactions from a run $\mathcal{C}$, we obtain a blockchain $\mathbf{B}_{\mathcal{C}}$.

We relate the computational and the symbolic models through a relation between symbolic runs $\mathcal{S}$ and computational runs $\mathcal{C}$: intuitively, $\mathcal{S}$ is *coherent* with $\mathcal{C}$ when each step in $\mathcal{S}$ is simulated by a step in $\mathcal{C}$. In the rest of this section we provide the main intuition about the coherence relation, relegating the full details to [13]. The coherence relation, denoted as $\sim$, is parameterized over two injective functions *txout* and *tkid* which track, respectively, the *names* x and the *tokens* τ occurring in $\Gamma_{\mathcal{S}}$, mapping them to transaction outputs (T, i), where T occurs in $\mathcal{C}$. We abbreviate $\mathcal{S} \sim_{txout,tkid} \mathcal{C}$ as $\mathcal{S} \sim \mathcal{C}$.

We simulate each symbolic token action in Figure 5 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 e_{TOK}, 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 (we display the complete script in Figure 10). 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 arg sequence by name rather than by index, e.g. we write o.op rather than o.arg.1.

Gen: We implement the *gen* action by the following script:

```
1 if not verrec(ptxo(1))          // ctxo is a gen
2 then ctxo.tkid = txid(ptxo(1)) // token id
3      and ptxo(1).val = 0        // spent txo has 0 BTC
4      and outlen(ctxo) = 1       // gen has 1 out
5      and ctxo.tkval > 0         // positive token val
6 else ... // the other branches must preserve token id
```

Recall that *gen* produces a symbolic step of the form:

$$\langle \mathsf{A}, 0 : \text{Ƀ} \rangle_x \xrightarrow{gen(x,v)} \langle \mathsf{A}, v : \tau \rangle_y \qquad (v > 0)$$

To translate this symbolic action into a computational one, we must spend a transaction output corresponding to $\langle \mathsf{A}, 0 : \text{Ƀ} \rangle_x$, and produce a fresh output corresponding to $\langle \mathsf{A}, v : \tau \rangle_y$. Assuming that the deposit x corresponds to an unspent transaction output $(\mathsf{T}', 1)$ on the blockchain, this requires to append a transaction T redeeming $(\mathsf{T}', 1)$, and ensuring that:

1) the parent transaction output $(\mathsf{T}', 1)$ is not a token deposit, but just a plain Ƀ deposit (line 1);

2) tkid is the identifier of the parent tx output (line 2). This corresponds to identifying the fresh name τ with the deposit name x of the redeemed ₿ deposit;
3) 0₿ are redeemed from the parent transaction (line 3);
4) T has exactly one output (line 4);
5) tkval 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 $(\mathsf{T}, 1)$, and ptxo(1) to $(\mathsf{T}', 1)$. 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 *join* and *xchg* actions.

Burn: We implement the *burn* action by the script:

```
1 versig(ctxo.owner,rtx.wit) and // check owner
2 verscr(false,rtxo(1)) and      // make rtx unspendable
3 inlen(rtxo(1)) = 1 and         // rtx has 1 in
4 outlen(rtxo(1)) = 1            // rtx has 1 out
```

Recall that *burn* produces a symbolic step:

$$\left(\|_{i \in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \xrightarrow{burn(x_1 \cdots x_n, y)} \mathbf{0}$$

There are two cases, according to whether we are burning a single token deposit, or one or more ₿ deposits. In the first case, assuming that the computational counterpart of x_1 is the output $(\mathsf{T}', 1)$, the corresponding computational step is to append a transaction T redeeming $(\mathsf{T}', 1)$. The witness of T must carry a signature of the owner A, and its output script is *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 *false* script.

Split: We implement the *split* action by the script:

```
1  versig(ctxo.owner,rtx.wit)              // check owner
2  and verrec(rtxo(1))                  // covenants on rtx
3  and verrec(rtxo(2))
4  and inlen(rtxo(1)) = 1                  // rtx has 1 in
5  and outlen(rtxo(1)) = 2                // rtx has 2 outs
6  and rtxo(1).tkval >= 0           // positive token value
7  and rtxo(2).tkval >= 0
8  and rtxo(1).owner = ctxo.owner       // preserve owner
9  and rtxo(1).tkid = ctxo.tkid          // preserve tkid
10 and rtxo(2).tkid = ctxo.tkid
11 and rtxo(1).tkval + rtxo(2).tkval = ctxo.tkval
```

Recall that *split* produces a symbolic step of the form:

$$\langle \mathsf{A}, (v + v') : \tau \rangle_x \xrightarrow{split(x, v, \mathsf{B})} \langle \mathsf{A}, v : \tau \rangle_y \mid \langle \mathsf{B}, v' : \tau \rangle_z$$

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

1) the witness of T carries a signature of the owner (line 1);
2) T has only one input and two outputs, both containing the same script of $(\mathsf{T}', 1)$ (line 2-5);
3) the tkval of T's outputs rtxo(1) and rtxo(2) are ≥ 0 (line 6-7), corresponding to the precondition $v, v' \geq 0$ in [Split];
4) tkid of T's outputs is the same of $(\mathsf{T}', 1)$ (line 9-10);
5) the sum of token values tkval of the outputs of T′ is equal to the token value of $(\mathsf{T}', 1)$ (line-11).

Join: We implement the *join* action by the script:

```
1  inlen(rtxo(1)) = 2                    // rtx has 2 ins
2  and outlen(rtxo(1)) = 1               // rtx has 1 out
3  and verrec(rtxo(1))                 // coventant on rtx
4  and verrec(stxo(2))         // covenants on both inputs
5  and verrec(stxo(1))
6  and ctxo.tkid = rtxo(1).tkid      // preserve token id
7  and versig(ctxo.owner, rtx.wit) // check sig of owner
8  and rtxo(1).tkval = stxo(1).tkval + stxo(2).tkval
```

Recall that *join* produces a symbolic step of the form:

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

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

1) T has two inputs and one output, containing the same script of $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (line 1-5);
2) the token identifier tkid of the output of T′ (rtxo(1)) is the same of $(\mathsf{T}', 1)$ (line 6);
3) the witness of T carries a signature of the owner (line 7);
4) the sum of token values tkval of both inputs is equal to the token value of $(\mathsf{T}, 1)$ (line 8).

Note that the script in $(\mathsf{T}', 1)$ ensures that the one in $(\mathsf{T}'', 1)$ 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:

```
1  inlen(rtxo(1)) = 2                    // rtx has 2 ins
2  and outlen(rtxo(1)) = 2               // rtx has 2 outs
3  and verrec(stxo(1))             // covenant on input 1
4  and verrec(rtxo(1))              // covenant on rtx(1)
5  and versig(ctxo.owner, rtx.wit)       // check  owner
6  and rtxo(1).owner = stxo(2).owner   // exchange owner
7  and rtxo(2).owner = stxo(1).owner
8  and rtxo(1).tkval = stxo(1).tkval   // preserve value
9  and rtxo(1).tkid = stxo(1).tkid      // preserve tkid
10 if verrec(stxo(2)) then        // exchange token/token
11       verrec(rtxo(2))           // covenant on rtx(2)
12   and rtxo(2).tkval = stxo(2).tkval // preserve tkval
13   and rtxo(2).tkid = stxo(2).tkid    // preserve tkid
14 else                            // exchange token/BTC
15       verscr(versig(ctxo.owner, rtx.wit), rtxo(2))
16   and rtxo(2).val = stxo(2).val       // preserve BTC
```

Recall that the symbolic *xchg* step has the form:

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

where τ must be a token, while τ' is either a ₿ or a token.

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

1) T has two inputs and two output (lines 1-2);
2) the first input and the first output of T must contain the same script of $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (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 ($\mathsf{rtxo}(1)$) must be equal to the owner in the second input $(\mathsf{T}'', 1)$ (line 6);
5) dually, the owner in the second output of T ($\mathsf{rtxo}(2)$) must be equal to the owner in the first input $(\mathsf{T}', 1)$ (line 7);
6) the token value and identifier of the first output of T ($\mathsf{rtxo}(1)$) must be equal to those of $(\mathsf{T}', 1)$ (line 8-9).

Furthermore, if $\mathsf{verrec}(\mathsf{stxo}(2))$ 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 $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (line 11);
2) the token value and identifier of the second output of T ($\mathsf{rtxo}(2)$) must be equal to those of $(\mathsf{T}'', 1)$ (line 12-13);

When exchanging a token with a ₿ deposit, we require:

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

Give: We implement the *give* action by the script:

```
1 inlen(rtxo(1)) = 1                    // rtx has 1 in
2 and outlen(rtxo(1)) = 1               // rtx has 1 out
3 and versig(ctxo.owner, rtx.wit)       // check  owner
4 and verrec(rtxo(1))                   // covenant on rtx(1)
5 and rtxo(1).tkid = ctxo.tkid          // preserve tkid
6 and rtxo(1).tkval = ctxo.tkval        // preserve value
```

Recall that *give* produces a symbolic step of the form:

$$\langle \mathsf{A}, v : \tau \rangle_x \xrightarrow{give(x,\mathsf{B})} \langle \mathsf{B}, v : \tau \rangle_y$$

Assuming that x corresponds to an unspent transaction output $(\mathsf{T}', 1)$, performing this step in Bitcoin requires to append a transaction T redeeming $(\mathsf{T}', 1)$, 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 ($\mathsf{rtxo}(1)$) contains the same script of $(\mathsf{T}', 1)$ (line 4);
4) the token value and identifier of the first output of T ($\mathsf{rtxo}(1)$) is the same of those of $(\mathsf{T}', 1)$ (lines 5-6).

Authorizations: Symbolic authorization steps (Figure 6) correspond to computational broadcasts of signatures. Computational users, however, can also broadcast other messages: in particular, adversaries can broadcast any arbitrary bit-string 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 $\mathsf{A}_1 \cdots \mathsf{A}_n$ own some bitcoins, represented in the symbolic run as deposits $\langle \mathsf{A}_i, v_i : ₿ \rangle_{x_i}$ for $i \in 1..n$, and in the computational run as transaction outputs $\mathsf{T}_{x_1} \cdots \mathsf{T}_{x_n}$, each one redeemable with A_i's private key. The dishonest users can sign an *arbitrary* T' which redeems all the $\mathsf{T}_{x_1} \cdots \mathsf{T}_{x_n}$, and append it to the blockchain. Crucially, T' may not correspond to *gen*, *split*, *join*, *xchg* or *give* actions. In this case, to obtain coherence, we simulate the appending of T' with a (previously authorized) *burn* of the deposits $\langle \mathsf{A}_i, v_i : ₿ \rangle_{x_i}$. 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 $\langle \mathsf{A}, v : \tau \rangle_x$ with a user-generated token τ is similar: appending T' is simulated by a symbolic *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 verrec and 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_{TOK} is a non-standard script, it could be used in a standard P2SH transaction, as in [22], if it did not exceed the 520-bytes limit. Taproot [23] 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 $\mathcal{S}$ have a corresponding unspent transaction output in any computational run $\mathcal{C}$ coherent with $\mathcal{S}$.

Lemma 3. *Let $\mathcal{S} \sim \mathcal{C}$. For each deposit $\langle \mathsf{A}, v : \tau \rangle_x$ in $\Gamma_{\mathcal{S}}$, there exists a distinct tx output $txout(x)$ in $UTXO(\mathbf{B}_{\mathcal{C}})$ 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 τ is minted in the symbolic world, a computational adversary creates a descendent T' of T with the token script and $\mathsf{tkid} = \tau$, effectively forging new units of τ. 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 $\mathcal{S} \sim \mathcal{C}$. Let (T, i) be a tx output in $UTXO(\mathbf{B}_\mathcal{C})$ storing $v : \tau$, with $\tau \neq ₿$. If $(\mathsf{T}, i).\mathsf{tkid}$ does not correspond to any tx output in $\mathbf{B}_\mathcal{C}$, then (T, i) is unspendable. Otherwise, if $(\mathsf{T}, i).\mathsf{tkid}$ occurs in $\mathbf{B}_\mathcal{C}$, and $(\mathsf{T}, i).\mathsf{tkid}$ is equal to $txout(x)$ for some x such that $gen(x, v')$ is fired in $\mathcal{S}$, then:*

1) *if $(\mathsf{T}, i) \notin \mathrm{ran}\,(txout)$ then (T, i) is unspendable;*
2) *if $(\mathsf{T}, i) = txout(y)$ for some y, then (T, i) is spendable, and $\langle \mathsf{A}, v : \tau \rangle_y$ occurs in $\Gamma_\mathcal{S}$, where $\mathsf{A} = (\mathsf{T}, i).\mathsf{owner}$.*

Note that all the results above require as hypothesis that $\mathcal{S} \sim \mathcal{C}$. In the rest of this section we show that, with overwhelming probability, for each computational run $\mathcal{C}$ there exists a symbolic run $\mathcal{S}$ which is coherent with $\mathcal{C}$. Actually, our computational soundness result is more precise than that. First, we consider only a subset Hon of users to be honest, while we consider all the others dishonest: without loss of generality, we model them as a single adversary 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 Σ^s_A 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. Σ^s_A receives as input a finite symbolic run $\mathcal{S}$, and outputs a finite set of enabled actions, with the constraint that Σ^s_A cannot output authorizations for $\mathsf{B} \neq \mathsf{A}$. We require strategies to be *persistent*: if on a run Σ^s_A chooses an action α, and α is not taken as the next step in the run (e.g., because some other user acts earlier), then Σ^s_A must still choose α after that step, if still enabled. The *adversary* Adv acts on behalf of all the dishonest users, and controls the scheduling among all users (including the honest ones). Her symbolic strategy Σ^s_Adv 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 Σ^s_Adv is a single action α (to be appended to the current run). To rule out authorization forgeries, we require that if α is an authorization by some honest A, then it must be chosen by Σ^s_A. Fixing a set of strategies $\mathbf{\Sigma}^s$ — both for the honest users and for the adversary — we obtain a unique run, which is made by the sequence of actions chosen by Σ^s_Adv when taking as input the outputs of the honest users' strategies. We say that this run is *conformant* to $\mathbf{\Sigma}^s$. When $\Sigma^s_\mathsf{Adv} \notin \mathbf{\Sigma}^s$, we say that $\mathcal{S}$ conforms to $\mathbf{\Sigma}^s$ when there exists some Σ^s_Adv such that $\mathcal{S}$ conforms to $\mathbf{\Sigma}^s \cup \{\Sigma^s_\mathsf{Adv}\}$.

Computational strategies: A computational strategy Σ^c_A for a honest user A is a PPTIME algorithm which receives as input a computational run $\mathcal{C}$, and outputs a finite set of computational labels. The choice among these labels is controlled by Adv's strategy, specified below. We assume that each user knows her private key, and the public keys of all the other users. We impose a few sanity constraints: (i) we forbid A to impersonate another user; (ii) if the strategy outputs a transaction T, then T must be a valid update of the blockchain $\mathbf{B}_\mathcal{C}$ obtained from the run in input, and all the witnesses of T have already been broadcast in the run; (iii) finally, strategies must be persistent, similarly to the symbolic case. Adv's computational strategy Σ^c_Adv is a PPTIME algorithm taking as input a run $\mathcal{C}$ and the moves chosen by each honest user. The strategy gives as output a single computational label, to be appended to the run. We assume that Adv can impersonate any other user. Given a symbolic strategy Σ^s_A, we can obtain a computational strategy $\Sigma^c_\mathsf{A} = \aleph(\Sigma^s_\mathsf{A})$ by implementing the symbolic actions as the corresponding computational ones (this can be done using the same technique as in [24]). Given a set of computational strategies $\mathbf{\Sigma}^c$ — both for the honest users and for the adversary — we probabilistically generate a *conformant* computational run, made by the sequence of actions chosen by Σ^c_Adv when taking as input the outputs of the honest users' strategies.

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 $\mathbf{\Sigma}^s$ be a set of symbolic strategies for all $\mathsf{A} \in \mathsf{Hon}$. Let $\mathbf{\Sigma}^c$ be a set of computational strategies such that $\Sigma^c_\mathsf{A} = \aleph(\Sigma^s_\mathsf{A})$ for all $\mathsf{A} \in \mathsf{Hon}$, and including an arbitrary adversary strategy Σ^c_Adv. For each $k \in \mathbb{N}$ and security parameter η, we define the following experiment:*

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

Then, the experiment returns 1 with overwhelming probability w.r.t. the security parameter η.

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 $\mathcal{C}$ and transaction outputs t, let:

$$P_t = \left\{ t' \;\middle|\; \begin{array}{l} t' \in UTXO(\mathbf{B}_\mathcal{C}) \text{ and } t' \text{ is spendable and} \\ t'.\mathsf{tkid} = t \text{ and } t'.\mathsf{scr} = e_{TOK} \end{array} \right\}$$

We define the balance of the computational token t in $\mathcal{C}$ as:

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

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

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

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

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 is* within high-level languages for Bitcoin contracts, e.g. BitML [24], [25], simplifying the design of financial contracts which manage tokens. For instance, we could model as follows a basic zero-coupon bond [26] where an investor A pays upfront to a bank B 5 units of a token τ, and receives back 1฿ after a maturity date t:

$$\texttt{split}(5{:}\tau \to \texttt{withdraw}\ \mathsf{B} \mid 1{:}฿ \to \texttt{after}\ t : \texttt{withdraw}\ \mathsf{A})$$

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 [27], [28]. An alternative approach is to provide a native support for tokens: the works [29], [30] follow this approach, proposing an extension of the UTXO model used in the Cardano blockchain [31]. The work [32] 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 [33]. 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 [34] has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [11]). Currently, covenants are supported by Bitcoin Cash [35], a mainstream blockchain platform originated from a fork of Bitcoin.

The work [14] proposes a new opcode CheckOutputVerify to explicitly constrain the outputs of the redeeming transaction, while [16] 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 [14] and [16] can constrain the script of the redeeming transaction to contain the same covenant of the spent one, enabling recursive covenants similarly to our verrec(rtxo(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 [16]: in Theorem 2 we show that, using our neighbourhood covenants, Bitcoin can actually simulate counter machines, assuming unbounded integers.

The work [36] 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 [36] features registers, loops, complex data structures, and native tokens, making it quite distant from an implementation on Bitcoin. Compared to [36], 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 [15] 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 [15] only constrain the redeeming transaction, their implementation is simpler, since it does not require to extend the UTXO set with the parents. As a downside, the covenants of [15] do not allow to implement fungible tokens.

Acknowledgements: Massimo Bartoletti is partially supported by Convenzione Fondazione di Sardegna e Atenei Sardi project F74I19000900007 *"ADAM"*. Stefano Lande is supported by P.O.R. F.S.E. 2014-2020. Roberto Zunino is partially supported by MIUR PON *"Distributed Ledgers for Secure Open Communities"*.

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] "Epobc protocol specification," 2020, https://github.com/chromaway/ngcccbase/wiki/EPOBC_simple.
[5] "Openassets protocol," 2020, https://github.com/OpenAssets.
[6] "Colu website," 2020, https://www.colu.com/.
[7] "Counterpary website," 2020, http://counterparty.io/.
[8] M. Bartoletti, B. Bellomy, and L. Pompianu, "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. Dashjr, "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. Swambo, S. Hommel, B. McElrath, and B. Bishop, "Bitcoin covenants: Three ways to control the future," *CoRR*, vol. abs/2006.16714, 2020.
[13] M. Bartoletti, S. Lande, and R. Zunino, "Computationally sound Bitcoin tokens," *CoRR*, vol. abs/2010.01347, 2020. [Online]. Available: https://arxiv.org/abs/2010.01347
[14] M. Möser, I. Eyal, and E. G. Sirer, "Bitcoin covenants," in *Financial Cryptography Workshops*, ser. LNCS, vol. 9604. Springer, 2016, pp. 126–141.
[15] M. Bartoletti, S. Lande, and R. Zunino, "Bitcoin covenants unchained," in *ISoLA*, ser. LNCS, vol. 12478. Springer, 2020, pp. 25–42.
[16] R. O'Connor and M. Piekarska, "Enhancing Bitcoin transactions with covenants," in *Financial Cryptography Workshops*, ser. LNCS, vol. 10323. Springer, 2017.
[17] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of Bitcoin transactions," in *Financial Cryptography and Data Security*, ser. LNCS, vol. 10957. Springer, 2018, pp. 541–560.
[18] 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.
[19] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on Bitcoin," *Commun. ACM*, vol. 59, no. 4, pp. 76–84, 2016.
[20] 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.
[21] 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.
[22] "Balzac: Bitcoin abstract language, analyzer and compiler," https://blockchain.unica.it/balzac/, 2018.
[23] A. T. Pieter Wuille, Jonas Nick, "Taproot: SegWit version 1 spending rules," 2020, BIP 341, https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.
[24] M. Bartoletti and R. Zunino, "BitML: a calculus for Bitcoin smart contracts," in *ACM CCS*, 2018.
[25] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, "Developing secure Bitcoin contracts with BitML," in *ESEC/FSE*, 2019.
[26] 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.
[27] "ERC-20 token standard," 2015, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.
[28] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "EIP 721: ERC-721 non-fungible token standard," https://eips.ethereum.org/EIPS/eip-721.
[29] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. P. Jones, P. Vinogradova, P. Wadler, and J. Zahnentferner, "UTXO$_{\mathrm{ma}}$: UTXO with multi-asset support," in *ISoLA*, 2020, to appear.
[30] 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.
[31] 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.
[32] J. Zahnentferner, "Multi-currency ledgers," Cryptology ePrint Archive, Report 2020/895, 2020, https://eprint.iacr.org/2020/895.
[33] G. Maxwell, "CoinCovenants using SCIP signatures, an amusingly bad idea," 2013, https://bitcointalk.org/index.php?topic=278122.0.
[34] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable off-chain instant payments," 2015. [Online]. Available: https://lightning.network/lightning-network-paper.pdf
[35] R. Kalis, "Cashscript — writing covenants," 2019. [Online]. Available: https://cashscript.org/docs/guides/covenants/
[36] 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.

```
if not verrec(ptxo(1)) then                          // ctxo is a token generator
      ctxo.op = 0                                                  // gen action
  and ctxo.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 ctxo.tkval > 0                                   // positive token value
else true

and

if rtxo.op = 1 then            /******************** BURN ********************/
      versig(ctxo.owner,rtx.wit)                                   // check owner
  and verscr(false,rtxo(1))                            // make rtx unspendable
  and inlen(rtxo(1)) = 1                                           // rtx has 1 in
  and outlen(rtxo(1)) = 1                                         // rtx has 1 out

else if rtxo.op = 2 then      /******************** SPLIT ********************/
      versig(ctxo.owner,rtx.wit)                                   // check owner
  and verrec(rtxo(1))                                    // covenants on rtx
  and verrec(rtxo(2))
  and inlen(rtxo(1)) = 1                                           // rtx has 1 in
  and outlen(rtxo(1)) = 2                                        // rtx has 2 outs
  and rtxo(1).tkval >= 0                               // positive token value
  and rtxo(2).tkval >= 0
  and rtxo(1).owner = ctxo.owner                                   // preserve owner
  and rtxo(1).tkid = ctxo.tkid                                      // preserve tkid
  and rtxo(2).tkid = ctxo.tkid
  and rtxo(1).tkval + rtxo(2).tkval = ctxo.tkval                   // preserve value

else if rtxo.op = 3 then       /******************** JOIN ********************/
      inlen(rtxo(1)) = 2                                            // rtx has 2 in
  and outlen(rtxo(1)) = 1                                          // rtx has 1 out
  and verrec(rtxo(1))                                            // coventant on rtx
  and verrec(stxo(2))                                 // covenants on both inputs
  and verrec(stxo(1))
  and ctxo.tkid = rtxo(1).tkid                                  // preserve token id
  and versig(ctxo.owner, rtx.wit)                                  // check owner
  and rtxo(1).tkval = stxo(1).tkval + stxo(2).tkval               // preserve value

else if rtxo.op = 4 then       /******************** XCHG ********************/
      inlen(rtxo(1)) = 2                                           // rtx has 2 ins
  and outlen(rtxo(1)) = 2                                         // rtx has 2 outs
  and versig(ctxo.owner, rtx.wit)                                   // check owner
  and verrec(stxo(1))                                  // covenant on input 1
  and verrec(rtxo(1))                                   // covenant on rtx(1)
  and rtxo(1).owner = stxo(2).owner                                  // swap owners
  and rtxo(2).owner = stxo(1).owner
  and rtxo(1).tkval = stxo(1).tkval                                // preserve value
  and rtxo(1).tkid = stxo(1).tkid                                   // preserve tkid
  if verrec(stxo(2)) then                     /***** EXCHANGE TOKEN/TOKEN *****/
        verrec(rtxo(2))                                          // covenant on rtx(2)
    and rtxo(2).tkval = stxo(2).tkval                              // preserve tkval
    and rtxo(2).tkid = stxo(2).tkid                                // preserve tkid
  else                                          /***** EXCHANGE TOKEN/BTC *****/
        verscr(versig(ctxo.owner, rtx.wit), rtxo(2))
    and rtxo(2).val = stxo(2).val                                   // preserve BTC

else if rtxo.op = 5 then       /******************** GIVE ********************/
      inlen(rtxo(1)) = 1                                           // rtx has 1 in
  and outlen(rtxo(1)) = 1                                         // rtx has 1 out
  and versig(ctxo.owner, rtx.wit)                                   // check owner
  and verrec(rtxo(1))                                           // covenant on rtx(1)
  and rtxo(1).tkid = ctxo.tkid                                      // preserve tkid
  and rtxo(1).tkval = ctxo.tkval                                   // preserve value

else false
```

Fig. 10: Full token script.