Bitcoin covenants unchained

Massimo Bartoletti[1], Stefano Lande[1], Roberto Zunino[2]

[1] Università degli Studi di Cagliari, Cagliari, Italy
[2] Università degli Studi di Trento, Trento, Italy

Abstract. Covenants are linguistic primitives that extend the Bitcoin script language, allowing transactions to constrain the scripts of the redeeming ones. Advocated as a way of improving the expressiveness of Bitcoin contracts while preserving the simplicity of the UTXO design, various forms of covenants have been proposed over the years. A common drawback of the existing descriptions is the lack of formalization, making it difficult to reason about properties and supported use cases. In this paper we propose a formal model of covenants, which can be implemented with minor modifications to Bitcoin. We use our model to specify some complex Bitcoin contracts, and we discuss how to exploit covenants to design high-level language primitives for Bitcoin contracts.

1 Introduction

Bitcoin is a decentralised infrastructure to transfer cryptocurrency between users. The log of all the currency transactions is recorded in a public, append-only, distributed data structure, called blockchain. Bitcoin implements a model of computation called *Unspent Transaction Output (UTXO)*: each transaction holds an amount of currency, and specifies conditions under which this amount can be redeemed by a subsequent transaction, which spends the old one. Compared to the *account-based* model, implemented e.g. by Ethereum, the UTXO model does not require a shared mutable state: the current state is given just by the set of unspent transaction outputs on the blockchain. While, on the one hand, this design choice fits well with the inherent concurrency of transactions, on the other hand the lack of a shared mutable state substantially complicates leveraging Bitcoin to implement *contracts*, i.e. protocols which transfer cryptocurrency according to programmable rules.

The literature has shown that Bitcoin contracts support a surprising variety of use cases, including e.g. crowdfunding [1, 10], lotteries and other gambling games [7,10,17,20,30,32], contingent payments [13], micro-payment channels [10, 37], and other kinds of fair computations [9, 29]. Despite this apparent richness, the fact is that Bitcoin contracts cannot express most of the use cases that are mainstream in other blockchain platforms (e.g., decentralised finance). There are several factors that limit the expressiveness of Bitcoin contracts. Among them, the crucial one is the script language used to express the redeeming conditions within transactions. This language only features a limited set of logic, arithmetic,

and cryptographic operators, but its has no loops, and it cannot access parts of the spent and of the redeeming transaction.

Several extensions of the Bitcoin script language have been proposed, with the aim to improve the expressiveness of Bitcoin contracts, while adhering to the UTXO model. Among these extensions, *covenants* are a class of script operators that allow a transaction to constrain how its funds can be used by the redeeming transactions. Covenants may also be recursive, by requiring the script of the redeeming transaction to contain the same covenant of the spent one. As noted by [35], recursive covenants would allow to implement Bitcoin contracts that execute state machines, by appending transactions to trigger state transitions.

Although the first proposals of covenants date back at least to 2013 [31], and that they are supported by Bitcoin fork "Bitcoin Cash" [28], their inclusion into Bitcoin is still uncertain, mainly because of the extremely cautious approach to implement changes to Bitcoin [27]. Still, the emerging of Bitcoin layer-2 protocols, like e.g. the Lightning Network [37], has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [38, 40]), and by the incorporation of covenants in Liquid's extensions to Bitcoin Script [34].

Since the goal of the existing proposals is to show how implementing covenants would impact on the performance of Bitcoin, they describe covenants from a low-level, technical perspective. We believe that a proper abstraction and formalization of covenants would also be useful, as it would simplify reasoning on the behaviour of Bitcoin contracts and on their properties.

Contributions We summarise our main contributions as follows:

- we introduce a formal model of Bitcoin covenants, inspired by the informal, low-level presentation in [33].
- we use our formal model to specify complex Bitcoin contracts, which largely extend the set of use cases expressible in pure Bitcoin;
- we discuss how to exploit covenants in the design of high-level language primitives for Bitcoin contracts.

2 The pure Bitcoin

We start by illustrating the Bitcoin transaction model. To this purpose we adapt the formalization in [12], omitting the parts that are irrelevant for our subsequent technical development.

Transactions In its simplest form, a Bitcoin transaction allows a user to transfer cryptocurrency (the *bitcoins*, ₿) to someone else. For this to be possible, bitcoins must be created at first. This is obtained through *coinbase* transactions (i.e., the first transaction of each mined block), whose typical form is:

T_0
in: $\bot$
wit: $\bot$
out: $\{\mathsf{scr} : \mathsf{versig}(pk_A, \mathsf{rtx.wit}), \mathsf{val} : 1₿\}$

We identify T_0 as a coinbase transaction by its in field, which does not point to any other previous transaction on the blockchain (formally, we model this as the undefined value $\bot$). The out field contains a pair, whose first element is a *script*, and the second one is the amount of bitcoins that will be redeemed by a subsequent transaction which points to T_0 and satisfies its script. In particular, the script $\mathsf{versig}(pk_{\mathsf{A}}, \mathsf{rtx.wit})$ verifies the signature in the wit field of the redeeming transaction (rtx) against A's public key pk_{A}.

Assume that T_0 is on the blockchain, and that A wants to transfer 1฿ to B. To do this, A can append to the blockchain a new transaction, e.g.:

T_1
in: T_0
wit: $sig_{sk_{\mathsf{A}}}(\mathsf{T}_1)$
out: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{B}}, \mathsf{rtx.wit}), \mathsf{val} : 1฿\}$

The in field points to T_0, and the wit field contains A's signature on T_1 (but for the wit field itself). This witness makes the script within $\mathsf{T}_0.\mathsf{out}$ evaluate to true, hence the redemption succeeds, and T_0 is *spent*.

The transactions T_0 and T_1 above only use part of the features of Bitcoin. More in general, transactions can collect bitcoins from many inputs, and split them between many outputs; further, they can use more complex scripts, and specify time constraints. Following the formalization in [12], we represent transactions as records with the following fields:

- 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 the list of inputs. 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 $\{\mathsf{scr} : e, \mathsf{val} : v\}$, where e is a script, and v is a currency value.
- $\mathsf{absLock}$ is a value, indicating the first moment in time when the transaction can be added to the blockchain;
- $\mathsf{relLock}$ is a list of values, of the same length as the list of inputs. Intuitively, if the value at index i is n, the transaction can be appended to the blockchain only if at least n time units have passed since the input transaction at index i has been appended.

We let f range over transaction fields, and we use the standard dot notation to access the fields of a record. For a transaction output (T, i) and $\mathsf{f} \in \{\mathsf{scr}, \mathsf{val}\}$, we write $(\mathsf{T}, i).\mathsf{f}$ for $\mathsf{T}.\mathsf{out}(i).\mathsf{f}$. For uniformity, we assume that $\mathsf{absLock}$ is a list of unit length; we omit null values in $\mathsf{absLock}$ and $\mathsf{relLock}$. When graphically rendering transactions, we usually write $\mathsf{f}(1) : \ell_1 \cdots \mathsf{f}(n) : \ell_n$ for $\mathsf{f} : \ell_1 \cdots \ell_n$, or just $\mathsf{f} : \ell_1$ when $n = 1$ (as in T_0 and T_1 above). 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 $sig_{sk_{\mathsf{A}}}(\mathsf{T})$.

$$[\![v]\!]_{\mathsf{T},i} = v \qquad [\![e \circ e']\!]_{\mathsf{T},i} = [\![e]\!]_{\mathsf{T},i} \circ_\bot [\![e']\!]_{\mathsf{T},i} \quad (\circ \in \{+,-,=,<\})$$

$$[\![\boldsymbol{e}.e']\!]_{\mathsf{T},i} = [\![e_j]\!]_{\mathsf{T},i} \text{ if } \boldsymbol{e} = e_1 \cdots e_k,\ [\![e']\!]_{\mathsf{T},i} = j, \text{ and } 1 \leq j \leq k$$

$$[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_{\mathsf{T},i} = \mathit{if}\ [\![e_0]\!]_{\mathsf{T},i}\ \mathit{then}\ [\![e_1]\!]_{\mathsf{T},i}\ \mathit{else}\ [\![e_2]\!]_{\mathsf{T},i} \qquad [\![\mathsf{rtx.wit}]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{wit}(i)$$

$$[\![|e|]\!]_{\mathsf{T},i} = \mathit{size}([\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{H}(e)]\!]_{\mathsf{T},i} = H([\![e]\!]_{\mathsf{T},i})$$

$$[\![\mathsf{versig}(e_1 \cdots e_n, e'_1 \cdots e'_m)]\!]_{\mathsf{T},i} = \mathsf{ver}_{[\![e_1]\!]_{\mathsf{T},i} \cdots [\![e_n]\!]_{\mathsf{T},i}}([\![e'_1]\!]_{\mathsf{T},i} \cdots [\![e'_m]\!]_{\mathsf{T},i}, \mathsf{T}, i)$$

$$[\![\mathsf{absAfter}\ e : e']\!]_{\mathsf{T},i} = \mathit{if}\ \mathsf{T}.\mathsf{absLock} \geq [\![e]\!]_{\mathsf{T},i}\ \mathit{then}\ [\![e']\!]_{\mathsf{T},i}\ \mathit{else}\ \bot$$

$$[\![\mathsf{relAfter}\ e : e']\!]_{\mathsf{T},i} = \mathit{if}\ \mathsf{T}.\mathsf{relLock}(i) \geq [\![e]\!]_{\mathsf{T},i}\ \mathit{then}\ [\![e']\!]_{\mathsf{T},i}\ \mathit{else}\ \bot$$

Fig. 1: Semantics of Bitcoin scripts.

Scripts Bitcoin scripts are small programs written in a non-Turing equivalent language. Whoever provides a witness that makes the script evaluate to "true", can redeem the bitcoins retained in the associated (unspent) output. In our model, scripts are terms with the following syntax, where $\circ \in \{+,-,=,<\}$, and where we write sequences of scripts in bold notation:

$$\begin{array}{rcl} e & ::= & v \mid e \circ e \mid \boldsymbol{e}.e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{rtx.wit} \mid \\ & & |e| \mid \mathsf{H}(e) \mid \mathsf{versig}(\boldsymbol{e}, \boldsymbol{e'}) \mid \mathsf{absAfter}\ e : e \mid \mathsf{relAfter}\ e : e \end{array}$$

Besides values v and the basic arithmetic/logical operators, scripts feature operators to access the elements of a sequence ($\boldsymbol{e}.e$), to access the witnesses of the redeeming transaction ($\mathsf{rtx.wit}$), to compute the size $|e|$ of a bitstring and its hash $\mathsf{H}(e)$. The script $\mathsf{versig}(\boldsymbol{e}, \boldsymbol{e'})$ evaluates to true iff the sequence of signatures resulting from the evaluation of $\boldsymbol{e'}$ (say, of length m) is verified by using m out of the n keys resulting from the evaluation of $\boldsymbol{e}$. The expressions $\mathsf{absAfter}\ e : e'$ and $\mathsf{relAfter}\ e : e'$ define absolute and relative time constraints: they evaluate as e' if the constraints are satisfied, otherwise their semantics is undefined. We assume a basic type system which rules out ill-formed scripts.

We define in Figure 1 the semantics of scripts. The script evaluation function $[\![\cdot]\!]_{\mathsf{T},i}$ takes two parameters: T is the redeeming transaction, and i is the index of the redeeming input/witness. We denote with H a public hash function, with $\mathit{size}(n)$ the size (in bytes) of an integer n, and with ver a multi-signature verification function (see [12] for the definition of these semantic operators). All the operators are *strict*, i.e. they evaluate to $\bot$ if some of their operands is $\bot$. We use syntactic sugar for scripts, e.g. *false* denotes $1 = 0$, *true* denotes $1 = 1$, while $e\ \mathsf{and}\ e'$ denotes $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ \mathit{false}$, and $e\ \mathsf{or}\ e'$ denotes $\mathsf{if}\ e\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ e'$.

Blockchains We model a blockchain $\mathbf{B}$ as a sequence of transactions $\mathsf{T}_0 \cdots \mathsf{T}_n$. For simplicity, we abstract from the fact that Bitcoin groups transactions into time-stamped blocks, and we identify the time-stamp of a transaction with its position in the blockchain. We say that the j-th output of the transaction T_i in the blockchain is *spent* iff there exists some transaction $\mathsf{T}_{i'}$ in the blockchain (with $i' > i$) and some j' such that $\mathsf{T}_{i'}.\mathsf{in}(j') = (\mathsf{T}_i, j)$.

e	::=	$\cdots$	
		$\mid$ $\mathsf{ctxo}(e).\mathsf{f}$	access part of the current transaction
		$\mid$ $\mathsf{rtxo}(e).\mathsf{f}$	access part of the redeeming transaction
		$\mid$ outidx	index of the redeemed output
		$\mid$ inidx	index of the redeeming input
		$\mid$ $\mathsf{verscr}(e, e)$	covenant
		$\mid$ $\mathsf{verrec}(e)$	recursive covenant

Fig. 2: Extended Bitcoin scripts ($\mathsf{f} \in \{\mathsf{arg}, \mathsf{scr}, \mathsf{val}\}$).

A transaction T_n is *valid* with respect to the blockchain $\mathbf{B} = \mathsf{T}_0 \cdots \mathsf{T}_{n-1}$ whenever the following conditions hold:

1. for each input i of T_n, if $\mathsf{T}_n.\mathsf{in}(i) = (\mathsf{T}', j)$ then:
   (a) $\mathsf{T}' = \mathsf{T}_h$, for some $h < n$ (i.e., T' is one of the transactions in $\mathbf{B}$);
   (b) the j-th output of T_h is not spent in $\mathbf{B}$;
   (c) $[\![\mathsf{T}_h.\mathsf{out}(j)]\!]_{\mathsf{T}_n,i} = \mathit{true}$;
   (d) $n - h \geq \mathsf{T}_n.\mathsf{relLock}(i)$;
2. $n \geq \mathsf{T}_n.\mathsf{absLock}$;
3. the sum of the amounts of the inputs of T_n is greater or equal to the sum of the amount of its outputs (the difference between the amount of inputs and that of outputs is the *fee* paid to miners).

The Bitcoin consensus protocol ensures that each T_i in the blockchain (except the coinbase T_0) is valid with respect to the sequence of past transactions $\mathsf{T}_0 \cdots \mathsf{T}_{i-1}$.

3 Extending Bitcoin with covenants

To extend Bitcoin with covenants, we amend the transaction model of the previous section as follows:

- we add to each output a field $\mathsf{arg} : \boldsymbol{a}$, where $\boldsymbol{a}$ is a sequence of values. Intuitively, the extra field can be used to encode a state within transactions.
- we add script operators to access the outputs of the current transaction and of the redeeming one (by contrast, pure Bitcoin scripts can only access the whole redeeming transaction, but not its parts).
- we add script operators to check whether the output scripts in the redeeming transaction match a given script, or a given output of the current transaction (by contrast, in pure Bitcoin the redeeming transaction is only used when verifying signatures).

We extend the syntax of scripts in Figure 2, and in Figure 3 we define their semantics. As in pure Bitcoin, the script evaluation function takes as parameters the redeeming transaction T and the index i of the redeeming input/witness. From them, it is possible to infer the current transaction $\mathsf{T}' = \mathit{fst}(\mathsf{T}.\mathsf{in}(i))$,

$$[\![\mathsf{rtxo}(e).\mathsf{f}]\!]_{\mathsf{T},i} = (\mathsf{T}, [\![e]\!]_{\mathsf{T},i}).\mathsf{f} \qquad [\![\mathsf{ctxo}(e).\mathsf{f}]\!]_{\mathsf{T},i} = (\mathit{fst}\ \mathsf{T}.\mathsf{in}(i), [\![e]\!]_{\mathsf{T},i}).\mathsf{f}$$

$$[\![\mathsf{inidx}]\!]_{\mathsf{T},i} = i \qquad [\![\mathsf{outidx}]\!]_{\mathsf{T},i} = \mathit{snd}\ \mathsf{T}.\mathsf{in}(i)$$

$$[\![\mathsf{verscr}(e, e')]\!]_{\mathsf{T},i} = (\mathsf{T}, [\![e]\!]_{\mathsf{T},i}).\mathsf{scr} \equiv e' \qquad [\![\mathsf{verrec}(e)]\!]_{\mathsf{T},i} = (\mathsf{T}, [\![e]\!]_{\mathsf{T},i}).\mathsf{scr} \equiv \mathsf{T}.\mathsf{in}(i).\mathsf{scr}$$

Fig. 3: Semantics of extended scripts.

and the index $j = \mathit{snd}(\mathsf{T}.\mathsf{in}(i))$ of the redeemed output. The script $\mathsf{ctxo}(k).\mathsf{f}$ evaluates to the field f of the k-th output of the current transaction; similarly, $\mathsf{rtxo}(k).\mathsf{f}$ operates on the redeeming transaction. The symbols outidx and inidx evaluate, respectively, to the index of the redeemed output and to that of the redeeming input. The last two scripts specify covenants, in basic and recursive form. The basic covenant $\mathsf{verscr}(k, e')$ checks that the k-th output script of the redeeming transaction is syntactically equal to e' (note that e' is *not* evaluated). The recursive covenant $\mathsf{verrec}(k)$ checks that the k-th output of the redeeming transaction is syntactically equal to the redeemed output script.

4 Use cases

We illustrate the expressive power of our extension through a series of use cases, which, at the best of our knowledge, cannot be expressed in Bitcoin. We denote with U_A^v an unspent transaction output $\{\mathsf{arg} : \varepsilon, \mathsf{scr} : \mathsf{versig}(\mathsf{A}, \mathsf{rtx.wit}), \mathsf{val} : v\ddot{\mathrm{B}}\}$, where ε denotes the empty sequence (we will usually omit arg when empty).

4.1 Crowdfunding

Assume that a start-up Z wants to raise funds through a crowdfunding campaign. The target of the campaign is to gather at least $v\ddot{\mathrm{B}}$ by time t. The contributors want the guarantee that if this target is not reached, then they will get back their funds after the expiration date. The start-up wants to ensure that contributions cannot be retracted before time t, or once $v\ddot{\mathrm{B}}$ have been gathered.

We implement this use case without covenants, but just constraining the val field of the redeeming transaction. To fund the campaign, a contributor A_i publishes the transaction T_i in Figure 4 (left), which uses the following script:

$$CF = \begin{array}{l} (\mathsf{versig}(\mathsf{Z}, \mathsf{rtx.wit})\ \mathsf{and}\ \mathsf{rtxo}(1).\mathsf{val} \geq v)\ \ \mathsf{or} \\ \mathsf{absAfter}\ t : \mathsf{versig}(\mathsf{A}_i, \mathsf{rtx.wit}) \end{array}$$

This script is a disjunction between two conditions. The first condition allows Z to redeem the bitcoins deposited in this output, provided that the output at index 1 of the redeeming transaction pays at least $v\ddot{\mathrm{B}}$ (note that this constraint, rendered as $\mathsf{rtxo}(1).\mathsf{val} \geq v$, is not expressible in pure Bitcoin). The second condition allows A_i to get back her contribution after the expiration date t.

T_i
in: $\mathsf{U}_{\mathsf{A}_i}^{v_i}$
wit: $\cdots$
out: $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : CF,\ \mathsf{val} : v_i ฿\}$

T_Z
in: $(\mathsf{T}_1, 1) \cdots (\mathsf{T}_n, 1)$
wit: $sig_\mathsf{Z}(\mathsf{T}_\mathsf{Z}) \cdots sig_\mathsf{Z}(\mathsf{T}_\mathsf{Z})$
out: $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : \mathsf{versig}(\mathsf{Z}, \mathsf{rtx.wit}),\ \mathsf{val} : v' ฿\}$

Fig. 4: Transactions for the crowdfunding contract.

T_0
in: U_A^1
wit: $sig_\mathsf{A}(\mathsf{T}_0)$
out: $\{\mathsf{arg} : \mathsf{A},\ \mathsf{scr} : NFT,\ \mathsf{val} : 1 ฿\}$

T_1
in: $(\mathsf{T}_0, 1)$
wit: $sig_\mathsf{A}(\mathsf{T}_1)$
out: $\{\mathsf{arg} : \mathsf{B},\ \mathsf{scr} : NFT,\ \mathsf{val} : 1 ฿\}$

Fig. 5: A creates a token with T_0, and transfers it to B with T_1.

Once contributors have deposited enough funds (i.e., there are n transactions $\mathsf{T}_1, \ldots, \mathsf{T}_n$ with $v' = v_1 + \cdots v_n \geq v$), Z can get $v' ฿$ by appending T_Z to the blockchain. Note that, compared to the assurance contract in the Bitcoin wiki [1], ours offers more protection to the start-up. Indeed, while in [1] any contributor can retract her funds at any time, this is not possible here until time t.

4.2 Non-fungible tokens

A non-fungible token represents the ownership of a physical or logical asset, which can be transferred between users. Unlike fungible tokens (e.g., ERC-20 tokens in Ethereum [2]), where each token unit is interchangeable with every other unit, non-fungible ones have unique identities. Further, they do not support split and join operations, unlike fungible tokens.

We start by implementing a subtly flawed version of the non-fungible token. Consider the transactions in Figure 5, which use the following script:

$$NFT \ = \ \mathsf{versig}(\mathsf{ctxo}(1).\mathsf{arg}, \mathsf{rtx.wit}) \text{ and } \mathsf{verrec}(1) \text{ and } \mathsf{rtxo}(1).\mathsf{val} = 1$$

User A mints a token by depositing $1 ฿$ in T_0: to declare her ownership over the token, she sets $\mathsf{out}(1).\mathsf{arg}$ to her public key. To transfer the token to B, A appends the transaction T_1, setting its $\mathsf{out}(1).\mathsf{arg}$ to B's public key.

To spend T_0, the transaction T_1 must satisfy the conditions specified by the script NFT: (i) the wit field must contain the signature of the current owner; (ii) the script at index 1 must be equal to that at the same index in T_0; (iii) the output at index 1 must have $1 ฿$ value, to preserve the integrity of the token. Once T_1 is on the blockchain, B can transfer the token to another user, by appending a transaction which redeems T_1.

The script NFT has a design flaw, already spotted in [33]: we show how A can exploit this flaw in Figure 6. Suppose we have two unspent transactions: T_A and T'_A, both representing a token owned by A (in their first and only output). The transaction T_2 can spend both of them, since it complies with all the validity conditions: indeed, NFT only constrains the script in the first output of the

T_2
in: $(\mathsf{T}_\mathsf{A}, 1)$ $(\mathsf{T}'_\mathsf{A}, 1)$
wit: $sig_\mathsf{A}(\mathsf{T}_2)$ $sig_\mathsf{A}(\mathsf{T}_2)$
out(1): $\{\mathsf{arg} : \mathsf{A},\ \mathsf{scr} : \mathit{NFT},\ \mathsf{val} : 1฿\}$
out(2): $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : \mathsf{versig}(\mathsf{A}, \mathsf{rtx.wit}),\ \mathsf{val} : 1฿\}$

Fig. 6: A exploits the flaw to destroy a token, redeeming its value.

redeeming transaction, while the other outputs are only subject to the standard validity conditions (in particular, that the sum of their values does not exceed the value in input). Actually, T_2 destroys one of the two tokens, and removes the covenant from the other one.

To solve this issue, we can amend the NFT script as follows:

$$\mathit{NFT}' \ = \ \mathsf{versig}(\mathsf{ctxo}(\mathsf{outidx}).\mathsf{arg}, \mathsf{rtx.wit}) \text{ and } \mathsf{verrec}(\mathsf{inidx}) \text{ and } \mathsf{rtxo}(\mathsf{inidx}).\mathsf{val} = 1$$

The amended script correctly handles the case of a transaction which uses different outputs to store different tokens. NFT' uses $\mathsf{ctxo}(\mathsf{outidx}).\mathsf{arg}$, instead of $\mathsf{ctxo}(1).\mathsf{arg}$ in NFT, to ensure that, when redeeming a given output, the signature of the owner of the token at *that* output is checked. Further, NFT' uses $\mathsf{verrec}(\mathsf{inidx})$, instead of $\mathsf{verrec}(1)$ in NFT, to ensure that the covenant is propagated exactly to the transaction output which is redeeming that token (i.e., the one at index inidx). Notice that the amendment would make T_2 invalid: indeed, the script in $\mathsf{T}'_\mathsf{A}.\mathsf{out}(1)$ would evaluate to false:

$$\begin{aligned}
[\![\mathit{NFT}']\!]_{\mathsf{T}_2,2} &= [\![\mathsf{verrec}(\mathsf{inidx})]\!]_{\mathsf{T}_2,2} \ \wedge \ \cdots \\
&= (\mathsf{T}_2, [\![\mathsf{inidx}]\!]_{\mathsf{T}_2,2}).\mathsf{scr} \equiv \mathsf{T}_2.\mathsf{in}(2).\mathsf{scr} \ \wedge \ \cdots \\
&= (\mathsf{T}_2, 2).\mathsf{scr} \equiv (\mathsf{T}'_\mathsf{A}, 1).\mathsf{scr} \ \wedge \ \cdots \\
&= \mathsf{versig}(\mathsf{A}, \mathsf{rtx.wit}) \equiv \mathit{NFT}' \ \wedge \ \cdots \\
&= \mathit{false}
\end{aligned}$$

An alternative patch, originally proposed in [33], is to add a unique identifier id to each token, e.g. by amending the NFT script as follows:

$$\mathit{NFT} \text{ and } id = id$$

This allows to mint distinguishable tokens. For instance, if the tokens in T_A and T'_A are distinguishable, T_2 cannot redeem both of them.

4.3 Vaults

Transaction outputs are usually secured by cryptographic keys (e.g. through the script $\mathsf{versig}(pk_\mathsf{A}, \mathsf{rtx.wit})$). Whoever knows the corresponding private key (e.g., sk_A) can redeem such an output: in case of key theft, the legitimate owner is left without defence. Vault transactions, introduced in [33], are a technique to mitigate this issue, by allowing the legitimate owner to abort the transfer.

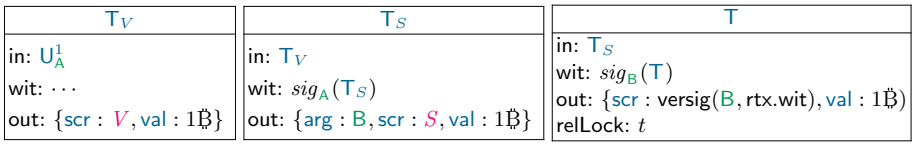

Fig. 7: Transactions for the basic vault.

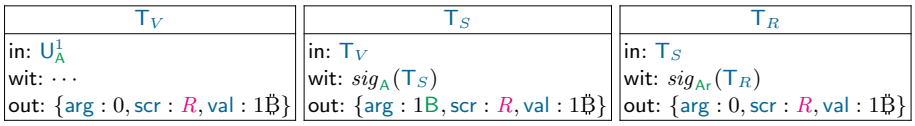

Fig. 8: Transactions for the recursive vault.

To create a vault, A deposits $1\ddot{\mathrm{B}}$ in a transaction T_V with the script V:

$$V = \mathsf{versig}(\mathsf{A}, \mathsf{rtx.wit}) \text{ and } \mathsf{verscr}(1, S)$$
$$S = \big(\mathsf{relAfter}\ t : \mathsf{versig}(\mathsf{ctxo}(\mathsf{outidx}).\mathsf{arg}, \mathsf{rtx.wit})\big) \text{ or } \mathsf{versig}(\mathsf{Ar}, \mathsf{rtx.wit})$$

The transaction T_V can be redeemed with the signature of A, but only by a *de-vaulting* transaction like T_S in Figure 7, which uses the script S. The output of the de-vaulting transaction T_S can be spent by the user set in its arg field, but only after a certain time t (e.g., by the transaction T in Figure 7). Before time t, A can cancel the transfer by spending T_S with her recovery key Ar.

A recursive vault The vault in Figure 7 has a potential issue, in that the recovery key may also be subject to theft. Although this issue is mitigated by hardware wallets (and by the infrequent need to interact with the recovery key), the vault modelled above does not discourage any attempt at stealing the key.

The issue can be solved by using a recursive covenant in the vault script R:

```
if ctxo(1).arg.1 = 0                                  // current state: vault
    then versig(A, rtx.wit) and verrec(1) and
    rtxo(1).arg.1 = 1                                 // next state: de-vaulting
else (relAfter t : versig(ctxo(1).arg.2, rtx.wit)) or // current state: de-vaulting
    versig(Ar, rtx.wit) and verrec(1) and
    rtxo(1).arg.1 = 0                                 // next state: vault
```

In this version of the contract, the vault and de-vaulting transactions (in Figure 8) have the same script. The first element of the arg sequence encodes the contract state (0 models the vault state, and 1 the de-vaulting state), while the second element is the user who can receive the bitcoin deposited in the vault. The recovery key Ar can only be used to append the re-vaulting transaction T_R, locking again the bitcoin into the vault.

Note that key theft becomes ineffective: indeed, even if both keys are stolen, the thief cannot take control of the bitcoin in the vault, as A can keep re-vaulting.

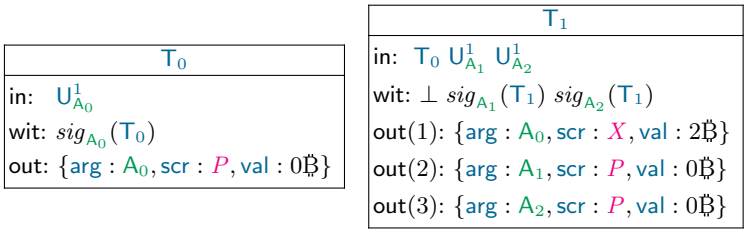

Fig. 9: Transactions for the pyramid scheme.

4.4 A pyramid scheme

Ponzi schemes are financial frauds which lure users under the promise of high profits, but which actually repay them only with the investments of new users. A pyramid scheme is a Ponzi scheme where the scheme creator recruits other investors, who in turn recruit other ones, and so on. Unlike in Ethereum, where several Ponzi schemes have been implemented as smart contracts [14,26], the limited expressive power of Bitcoin contract only allows for off-chain schemes [41].

We design the first "smart" pyramid scheme in Bitcoin using the transactions in Figure 9, where:

$$\begin{aligned} P &= \mathsf{verscr}(1, X)\ \mathsf{and}\ \mathsf{rtxo}(1).\mathsf{arg} = \mathsf{ctxo}(\mathsf{outidx}).\mathsf{arg}\ \mathsf{and}\ \mathsf{rtxo}(1).\mathsf{val} = 2 \\ &\quad\ \mathsf{and}\ \mathsf{verrec}(2)\ \mathsf{and}\ \mathsf{verrec}(3) \\ X &= \mathsf{versig}(\mathsf{ctxo}(\mathsf{outidx}).\mathsf{arg}, \mathsf{rtx.wit}) \end{aligned}$$

To start the scheme, a user A_0 deposits 1₿ in the transaction T_0 (we burn this bitcoin for uniformity, so that each user earns at most 1₿ from the scheme). To make a profit, A_0 must convince other two users, say A_1 and A_2, to join the scheme. This requires the cooperation of A_1 and A_2 to publish a transaction which redeems T_0. The script P ensures that this redeeming transaction has the form of T_1 in Figure 9, i.e. $\mathsf{out}(1)$ transfers 2₿ to A_0, while the scripts in $\mathsf{out}(2)$ and $\mathsf{out}(3)$ ensure that the same behaviour is recursively applied to A_1 and A_2.

Overall, the contract ensures that, as long as new users join the scheme, each one earns 1₿. Of course, as in any Ponzi scheme, at a certain point it will no longer be possible to find new users, so those at the leaves of the transaction tree will just lose their investment.

4.5 King of the Ether Throne

King of the Ether Throne [3] is an Ethereum contract, which has been popular for a while around 2016, until a bug caused its funds to be frozen. The contract is initiated by a user, who pays an entry fee v_0 to become the "king". Another user can usurp the throne by paying $v_1 = 1.5v_0$ fee to the old king, and so on until new usurpers are available. Of course this leads to an exponential growth of the fee needed to become king, so subsequent versions of the contract introduced mechanisms to make the current king die if not ousted within a certain time.

T_0	T_1
in: $\mathsf{U}_{\mathsf{A}_0}^{v_0}$	in: $(\mathsf{T}_0, 1)\ \mathsf{U}_{\mathsf{A}_1}^{v_1}$
wit: $sig_{\mathsf{A}_0}(\mathsf{T}_0)$	wit: $\bot\ sig_{\mathsf{A}_1}(\mathsf{T}_1)$
out(1): $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : K, \mathsf{val} : 0฿\}$	out(1): $\{\mathsf{arg} : \mathsf{A}_1, \mathsf{scr} : K, \mathsf{val} : 0฿\}$
out(2): $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : \mathsf{versig}(\mathsf{A}_0, \mathsf{rtx.wit}), \mathsf{val} : v_0฿\}$	out(2): $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : X, \mathsf{val} : v_1฿\}$

Fig. 10: Transactions for King of the Ether Throne.

Although the logic to distribute money substantially differs from that in Section 4.4, this is still an instance of Ponzi scheme, since investors are only paid with the funds paid by later investors.

We implement the original version of the contract, fixing the multiplier to 2 instead of 1.5, since Bitcoin scripts do not support multiplication. The contract uses the transactions in Figure 10 for the first two kings, A_0 and A_1, where:

$$\begin{aligned} K &= \mathsf{verrec}(1)\ \text{ and }\ \mathsf{rtxo}(2).\mathsf{arg} = \mathsf{ctxo}(1).\mathsf{arg}\ \text{ and } \\ &\qquad \mathsf{rtxo}(2).\mathsf{val} \geq \mathsf{ctxo}(2).\mathsf{val} + \mathsf{ctxo}(2).\mathsf{val}\ \text{ and } \mathsf{verscr}(2, X) \\ X &= \mathsf{versig}(\mathsf{ctxo}(2).\mathsf{arg}, \mathsf{rtx.wit}) \end{aligned}$$

We use the arg field in out(1) to record the new king, and that in out(2) for the old one. The clause $\mathsf{rtxo}(2).\mathsf{arg} = \mathsf{ctxo}(1).\mathsf{arg}$ in K preserves the old king in the redeeming transaction. The clause $\mathsf{rtxo}(2).\mathsf{val} \geq \mathsf{ctxo}(2).\mathsf{val} + \mathsf{ctxo}(2).\mathsf{val}$ ensures that his compensation is twice the value he paid. Finally, verscr guarantees that the old king can redeem his compensation via out(2).

5 Implementing covenants on Bitcoin

We now discuss how to implement covenants in Bitcoin, and their computational overhead. First, during the script verification, we need to access both the redeeming transaction and the one containing the output being spent. This can be implemented by adding a new data structure to store unspent or partially unspent transaction outputs, and modifying the entries of the UTXO set to link each unspent output to the enclosing transaction.

The language primitives that check the redeeming transaction script, verscr and verrec, can be implemented through an opcode similar to CheckOutputVerify described in [33]. While [33] uses placeholders to represent variable parts of the script, e.g., versig(<*pubKey*>, rtx.wit), we use operators to access the needed parts of a transaction, e.g., versig(ctxo(1).arg, rtx.wit). Thus, to check if two scripts are the same we just need to compare their hashes, while [33] needs to instantiate the placeholders. Similarly, we can use the hash of the script within verscr. The work [35] implements covenants without introducing operators to explicitly access the redeeming transaction. Instead, they exploit the current implementation of versig, which checks a signature on data that is build by implicitly accessing the redeeming transaction, to define a new operator CheckSigFromStack.

The arg part of each output can be stored at the beginning of the output script, without altering the structure of pure Bitcoin transactions. Similarly to the implementation of parameters in Balzac [6,12], the arguments are pushed on the alternative stack at the beginning of the script, then duplicated and copied in the main stack before the actual output script starts. Note that arguments need to be discharged when hashing the script for verrec/verscr. For this, it is enough to skip a known-length prefix of the script.

Even though the use cases in Section 4 extensively use non-standard scripts, they can be encoded as standard transactions using P2SH [5], as done in [6,12]. Crucially, the hash also covers the arg field, which is therefore not malleable.

6 Using covenants in high-level contract languages

As witnessed by the use cases in Section 4, crafting a contract at the level of Bitcoin transactions can be complex and error-prone. To simplify this task, the work [18] has introduced a high-level contract language, called BitML, with a secure compiler to pure Bitcoin transactions. BitML has primitives to `withdraw` funds from a contract, to `split` a contract (and its funds) into subcontracts, to request the authorization from a participant A before proceeding with a subcontract C (written $\mathsf{A} : C$), to postpone the execution of C after a given time t (written $\texttt{after}\, t : C$), to `reveal` committed secrets, and to branch between two contracts (written $C + C'$). A recent paper [16] extends BitML with a new primitive that allows participants to (consensually) renegotiate a contract, still keeping the ability to compile into pure Bitcoin.

Despite the variety of use cases shown in [11,15], BitML has known expressiveness limits, given by the requirement to have pure Bitcoin as its compilation target. For instance, BitML cannot specify recursive contracts (just as pure Bitcoin cannot), unless all participants agree to perform the recursive call [16]. In this section we discuss how to improve the expressiveness of BitML, assuming to use Bitcoin with covenants as compilation target. We illustrate our point by a couple of examples, postponing the formal treatment of this extended BitML and of its secure compilation to future work.

Covenants allow us to extend BitML with the construct:

$$?\boldsymbol{x}\ \texttt{if}\, b.\ \mathtt{X}(\boldsymbol{x})$$

Intuitively, the prefix $?\boldsymbol{x}\ \texttt{if}\, b$ can be fired whenever a participant provides a sequence of arguments $\boldsymbol{x}$ and makes the predicate b true. Once the prefix is fired, the contract proceeds as the continuation $\mathtt{X}(\boldsymbol{x})$, which will reduce according to the equation defining X.

Using this construct, we can model the "King of the Ether Throne" contract of Section 4.5 (started by A with an investment of 1₿) as $\mathtt{X}(\mathsf{A}, 1)$, where:

$$\begin{aligned} \mathtt{X}(\mathsf{a}, v) &= ?\mathsf{b}\ \texttt{if}\ \mathsf{val} \geq 2v.\ \mathtt{Y}(\mathsf{a}, \mathsf{b}, \mathsf{val}) \\ \mathtt{Y}(\mathsf{a}, \mathsf{b}, v) &= \texttt{split}\ \big(0 \to \mathtt{X}(\mathsf{b}, v) \mid v \to \texttt{withdraw}\ \mathsf{a}\big) \end{aligned}$$

```
1  def arg.1 = q      // state    0 = <X(A,-),1>
2                     // state    1 = <Y(a,b,v),v>
3                     // state    2 = <X(b,v),0> | <withdraw a,v>
4  def arg.2 = oldK   // old King
5  def arg.3 = newK   // new king
6  def arg.4 = v      // paid fee
7
8  verrec(1) and                                      // out(1) preserves covenant
9  if ctxo(1).q = 0 then                              // state 0
10       rtxo(1).q = 1                                // state transition 0 -> 1
11   and rtxo(1).oldK = ctxo(1).newK                  // usurp the throne
12   and rtxo(1).val >= ctxo(1).val + ctxo(1).val     // fee at least doubled
13   and rtxo(1).v = rtxo(1).val                      // instantiate v
14 else if ctxo(1).q = 1 then                         // state 1
15       rtxo(1).q = 2                                // state transition 1 -> 2
16   and rtxo(1).newK = ctxo(1).newK                  // preserve new king
17   and rtxo(1).v = ctxo(1).v                        // preserve v
18   and rtxo(1).val = 0                              // reset value in out(1)
19   and rtxo(2).oldK = ctxo(1).oldK                  // set old king
20   and verscr(2,versig(ctxo(2).oldK,rtx.wit))       // covenant to pay old king
21   and rtxo(2).val = ctxo(1).val                    // preserve value in out(2)
22 else if ctxo(1).q = 2 then                         // state 2
23       rtxo(1).q = 1                                // state transition 2 -> 1
24   and rtxo(1).oldK = ctxo(1).newK                  // usurp the throne
25   and rtxo(1).val >= ctxo(1).v + ctxo(1).v         // fee at least doubled
26   and rtxo(1).v = rtxo(1).val                      // update v
```

Fig. 11: Script for King of the Ether Throne, obtained by compiling BitML.

The contract $\mathtt{X}(\mathsf{a}, v)$ models a state where a is the current king, and v is his investment. The guard $\mathsf{val} \geq 2v$ becomes true when some participant injects funds into the contract, making its value (val) greater than $2v$. This participant can choose the value for b, i.e. the new king. The contract proceeds as $\mathtt{Y}(\mathsf{a}, \mathsf{b}, \mathsf{val})$, which has two parallel branches. The first branch makes $\mathsf{val}\,฿$ available to the old king; the second branch has zero value, and it reboots the game, recording the new king b and his investment.

A possible computation of A starting the scheme with $1฿$ is the following, where we represent a contract C storing $v฿$ as a term $\langle C, v฿ \rangle$:

$$
\begin{aligned}
\langle \mathtt{X}(\mathsf{A}, -), 1฿ \rangle &\to \langle \mathtt{Y}(\mathsf{A}, \mathsf{B}, 2), 2฿ \rangle && (\mathsf{B} \text{ pays } 2฿ \text{ fee}) \\
&\to \langle \mathtt{X}(\mathsf{B}, 2), 0฿ \rangle \mid \langle \mathtt{withdraw}\ \mathsf{A}, 2฿ \rangle && (\text{contract splits}) \\
&\to \langle \mathtt{X}(\mathsf{B}, 2), 0฿ \rangle \mid \langle \mathsf{A}, 2฿ \rangle && (\mathsf{A} \text{ redeems } 2฿) \\
&\to \langle \mathtt{Y}(\mathsf{B}, \mathsf{C}, 4), 4฿ \rangle \mid \langle \mathsf{A}, 2฿ \rangle && (\mathsf{C} \text{ pays } 4฿ \text{ fee}) \\
&\to \langle \mathtt{X}(\mathsf{C}, 4), 0฿ \rangle \mid \langle \mathtt{withdraw}\ \mathsf{B}, 4฿ \rangle \mid \langle \mathsf{A}, 2฿ \rangle && (\text{contract splits}) \\
&\to \langle \mathtt{X}(\mathsf{C}, 4), 0฿ \rangle \mid \langle \mathsf{B}, 4฿ \rangle \mid \langle \mathsf{A}, 2฿ \rangle && (\mathsf{B} \text{ redeems } 4฿)
\end{aligned}
$$

Executing a step of the BitML contract corresponds, in Bitcoin, to appending a transaction containing in $\mathsf{out}(1)$ the script in Figure 11. The script implements a state machine, using $\mathsf{arg}.1$ to record the current state, and the other parts of arg for the old king, the new king, and v. The $\mathsf{verrec}(1)$ at line 8 preserves the script in $\mathsf{out}(1)$. To pay the old king, we use the verscr at line 20, which constrains the script in $\mathsf{out}(2)$ of the transaction corresponding to the BitML state $\langle \mathtt{X}(\mathsf{b}, v), 0฿ \rangle \mid \langle \mathtt{withdraw}\ \mathsf{a}, v฿ \rangle$.

We now apply our extended BitML to specify a more challenging use case, i.e. a recursive coin-flipping game where two players A and B repeatedly flip coins, and the one who wins two consecutive flips takes the pot. The precondition to stipulate the contract requires each player to deposit $1\ddot{\mathrm{B}}$ as a bet. The game first makes each player commit to a secret, using a timed-commitment protocol [21]. The secrets are then revealed, and the winner of a flip is determined as a function of the two secrets. The game starts another flip if the current winner is different from that of the previous flip, otherwise the pot is transferred to the winner.

We model the recursive coin-flipping game as the (extended) BitML contract $\mathtt{X}_{\mathsf{A}}(\mathsf{C})$, where $\mathsf{C} \neq \mathsf{A}, \mathsf{B}$, using the following defining equations:

$$
\begin{aligned}
\mathtt{X}_{\mathsf{A}}(\mathsf{w}) &= \mathsf{A} : ?h_{\mathsf{A}} \,.\, \mathtt{X}_{\mathsf{B}}(\mathsf{w}, h_{\mathsf{A}}) \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B} \\
\mathtt{X}_{\mathsf{B}}(\mathsf{w}, h_{\mathsf{A}}) &= \mathsf{B} : ?h_{\mathsf{B}} \,.\, \mathtt{Y}_{\mathsf{A}}(\mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}}) \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A} \\
\mathtt{Y}_{\mathsf{A}}(\mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}}) &= ?s_{\mathsf{A}}\ \mathtt{if}\ H(s_{\mathsf{A}}) = h_{\mathsf{A}} \,.\, \mathtt{Y}_{\mathsf{B}}(\mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}}) \\
&\quad + \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B} \\
\mathtt{Y}_{\mathsf{B}}(\mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}}) &= ?s_{\mathsf{B}}\ \mathtt{if}\ H(s_{\mathsf{B}}) = h_{\mathsf{B}}\ \mathtt{and}\ 0 \leq s_{\mathsf{B}} \leq 1 \,.\, \mathtt{W}(\mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}}) \\
&\quad + \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A} \\
\mathtt{W}(\mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}}) &= \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathtt{and}\ \mathsf{w} = \mathsf{A} : \mathtt{withdraw}\ \mathsf{A} && \text{// A won twice} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathtt{and}\ \mathsf{w} \neq \mathsf{A} : \mathtt{X}_{\mathsf{A}}(\mathsf{A}) && \text{// A won last flip} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathtt{and}\ \mathsf{w} = \mathsf{B} : \mathtt{withdraw}\ \mathsf{B} && \text{// B won twice} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathtt{and}\ \mathsf{w} \neq \mathsf{B} : \mathtt{X}_{\mathsf{A}}(\mathsf{B}) && \text{// B won last flip}
\end{aligned}
$$

The contract $\mathtt{X}_{\mathsf{A}}(\mathsf{w})$ models a state where w is the last winner, and A must commit to her secret. To do that, A must authorize an input h_{A}, which represents the hash of her secret. If A does not commit within t, then the pot can be redeemed by B as a compensation (here, the primitive $\mathtt{afterRel}\, t : C$ models a relative timeout). Similarly, $\mathtt{X}_{\mathsf{B}}(\mathsf{w})$ models B's turn to commit. In $\mathtt{Y}_{\mathsf{A}}(\mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}})$, A must reveal her secret s_{A}, or otherwise lose her deposit. The contract $\mathtt{Y}_{\mathsf{B}}(\mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}})$ is the same for B, except that here we additionally check that B's secret is either 0 or 1 (this is needed to ensure fairness, as in the two-player lottery in [18]). The flip winner is A if the secrets of A and B are equal, otherwise it is B. If the winner is the same as the previous round, the winner can withdraw the pot, otherwise the game restarts, recording the last winner.

This coin flipping game is fair, i.e. the expected payoff of a *rational* player is always non-negative, notwithstanding the behaviour of the other player.

7 Conclusions and future work

We have proposed a formalisation of Bitcoin covenants, and we have exploited it to present a series of use cases which appear to be unfeasible in pure Bitcoin. We have introduced high-level contract primitives that exploit covenants to enable recursion, and allow contracts to receive new funds and parameters at runtime.

Known limitations Most of the scripts crafted in our use cases would produce non-standard transactions, that are rejected by Bitcoin nodes. To produce standard transactions from non-standard scripts, we can exploit P2SH [5]. This requires the transaction output to commit to the hash of the script, while the actual script is revealed in the witness of the redeeming transaction. Since, to check its hash, the script needs to be pushed to the stack, and the maximum size of a stack element is 520 bytes, longer scripts would be rejected. This clearly affects the expressiveness of contracts, as already observed in [11]. In particular, since the size of a script grows with the number of contract states (see e.g. Figure 11), contracts with many states would easily violate the 520 bytes limit. The introduction of Taproot [36] would mitigate this limit. 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. Another expressiveness limit derives from the fact that covenants can only constrain the scripts of the redeeming transaction. While this is enough to express non-fungible tokens (see Section 4.2), fungible ones seem to require more powerful mechanisms, because of the join operation. An alternative technique to enhancing covenants is to implement fungible tokens natively [24, 25], or to enforce their logic through a sidechain [34].

Verification Although designing contracts in the UTXO model seems to be less error-prone than in the shared memory model, e.g. because of the absence of reentrancy vulnerabilities (like the one exploited in the Ethereum DAO attack [4]), Bitcoin contracts may still contain security flaws. Therefore, it is important to devise verification techniques to detect security issues that may lead to the theft or freezing of funds. Recursive covenants make this task harder than in pure Bitcoin, since they can encode infinite-state transition systems, as in most of our use cases. Hence, model-checking techniques based on the exploration of the whole state space, like the one used in [8], cannot be applied.

High-level Bitcoin contracts The compiler of our extension of BitML is just sketched in Section 6, and we leave as future work its formal definition, as well as the extension of the computational soundness results of [18], ensuring the correspondence between the symbolic semantics of BitML and the underlying computational level of Bitcoin. Continuing along this line of research, it would be interesting to study new linguistic primitives that fully exploit the expressiveness of Bitcoin covenants, and to extend accordingly the verification technique of [19]. Note that our extension of the UTXO model is more restrictive than the one in [23], as the latter abstracts from the script language, just assuming that scripts denote any pure functions [42]. This added flexibility can be exploited to design expressive high-level contract languages like Marlowe [39] and Plutus [22].

Acknowledgements Massimo Bartoletti is partially supported by Aut. Reg. of Sardinia project "Sardcoin". Stefano Lande is partially supported by P.O.R. F.S.E. 2014-2020. Roberto Zunino is partially supported by MIUR PON 2018 "Distributed Ledgers for Secure Open Communities" ARS01_00587.

References

1. Bitcoin wiki - contracts - assurance contracts. https://en.bitcoin.it/wiki/Contract#Example_3:_Assurance_contracts (2012)
2. ERC-20 token standard (2015), https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
3. King ot the Ether Throne (2016), https://web.archive.org/web/20160211005112/https://www.kingoftheether.com/
4. Understanding the DAO attack (June 2016), http://www.coindesk.com/understanding-dao-hack-journalists/
5. Bitcoin wiki - Pay-to-Script Hash. https://en.bitcoinwiki.org/wiki/Pay-to-Script_Hash (2017)
6. Balzac: Bitcoin abstract language, analyzer and compiler. https://blockchain.unica.it/balzac/ (2018)
7. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via Bitcoin deposits. In: Financial Cryptography Workshops. LNCS, vol. 8438, pp. 105–121. Springer (2014). https://doi.org/10.1007/978-3-662-44774-1_8
8. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Modeling Bitcoin contracts by timed automata. In: FORMATS. LNCS, vol. 8711, pp. 7–22. Springer (2014). https://doi.org/10.1007/978-3-319-10512-3_2
9. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P. pp. 443–458 (2014). https://doi.org/10.1109/SP.2014.35
10. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: SoK: unraveling Bitcoin smart contracts. In: POST. LNCS, vol. 10804, pp. 217–242. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6
11. Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., Zunino, R.: Developing secure Bitcoin contracts with BitML. In: ESEC/FSE (2019). https://doi.org/https://doi.org/10.1145/3338906.3341173
12. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Financial Cryptography and Data Security. LNCS, vol. 10957. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6
13. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS. LNCS, vol. 9879, pp. 261–280. Springer (2016). https://doi.org/10.1007/978-3-319-45741-3_14
14. Bartoletti, M., Carta, S., Cimoli, T., Saia, R.: Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. Future Gener. Comput. Syst. **102**, 259–277 (2020). https://doi.org/10.1016/j.future.2019.08.014
15. Bartoletti, M., Cimoli, T., Zunino, R.: Fun with Bitcoin smart contracts. In: ISoLA. pp. 432–449 (2018). https://doi.org/10.1007/978-3-030-03427-6_32
16. Bartoletti, M., Murgia, M., Zunino, R.: Renegotiation and recursion in Bitcoin contracts. In: COORDINATION. LNCS, vol. 12134, pp. 261–278. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_17
17. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0
18. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. In: ACM CCS (2018). https://doi.org/10.1145/3243734.3243795

19. Bartoletti, M., Zunino, R.: Verifying liquidity of Bitcoin contracts. In: POST. LNCS, vol. 11426, pp. 222–247. Springer (2019)
20. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014). https://doi.org/10.1007/978-3-662-44381-1_24
21. Boneh, D., Naor, M.: Timed commitments. In: CRYPTO. LNCS, vol. 1880, pp. 236–254. Springer (2000). https://doi.org/10.1007/3-540-44598-6
22. Brünjes, L., Gabbay, M.J.: UTxO- vs account-based smart contract blockchain programming paradigms. CoRR **abs/2003.14271** (2020)
23. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: Financial Cryptography Workshops (2020), to appear
24. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P.: Native custom tokens in the extended UTXO model. In: ISoLA (2020), to appear
25. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P., Zahnentferner, J.: UTXO_{ma}: UTXO with multi-asset support. In: ISoLA (2020), to appear
26. Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P., Zhou, Y.: Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology. In: WWW. pp. 1409–1418. ACM (2018). https://doi.org/10.1145/3178876.3186046
27. Dashjr, L.: BIP 0002 (2016), https://en.bitcoin.it/wiki/BIP_0002
28. Kalis, R.: Cashscript — writing covenants (2019), https://cashscript.org/docs/guides/covenants/
29. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014). https://doi.org/10.1145/2660267.2660380
30. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015). https://doi.org/10.1145/2810103.2813712
31. Maxwell, G.: CoinCovenants using SCIP signatures, an amusingly bad idea (2013), https://bitcointalk.org/index.php?topic=278122.0
32. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops. pp. 4–13 (2017). https://doi.org/10.1109/EuroSPW.2017.44
33. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography Workshops. LNCS, vol. 9604, pp. 126–141. Springer (2016). https://doi.org/10.1007/978-3-662-53357-4_9
34. Nick, J., Poelstra, A., Sanders, G.: Liquid: a Bitcoin sidechain. https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf (2020)
35. O'Connor, R., Piekarska, M.: Enhancing Bitcoin transactions with covenants. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0_12
36. Pieter Wuille, Jonas Nick, A.T.: Taproot: SegWit version 1 spending rules (2020), BIP 341, https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki
37. Poon, J., Dryja, T.: The Bitcoin Lightning Network: Scalable off-chain instant payments (2015), https://lightning.network/lightning-network-paper.pdf
38. Rubin, J.: CHECKTEMPLATEVERIFY (2020), BIP 119, https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki
39. Seijas, P.L., Thompson, S.J.: Marlowe: Financial contracts on blockchain. In: ISoLA. LNCS, vol. 11247, pp. 356–375. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_27

40. Swambo, J., Hommel, S., McElrath, B., Bishop, B.: Bitcoin covenants: Three ways to control the future. CoRR **abs/2006.16714** (2020)
41. Vasek, M., Moore, T.: There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams. In: Financial Cryptography and Data Security. pp. 44–61 (2015). https://doi.org/10.1007/978-3-662-47854-7_4
42. Zahnentferner, J.: An abstract model of UTxO-based cryptocurrencies with scripts. Cryptology ePrint Archive **2018/469** (2018), https://eprint.iacr.org/2018/469