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 transactions 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, 9], lotteries and other gambling games [6, 9, 14, 17, 23, 25], contingent payments [12], micro-payment channels [9, 29], and other kinds of fair computations [8, 22]. 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). Indeed, 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 [27], 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 [24], their inclusion into the official Bitcoin protocol is still uncertain, mainly because of the extremely cautious approach to implement changes to Bitcoin [21]. Still, the emerging of Bitcoin layer-2 protocols, like e.g. the Lightning Network [29], has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [30]).

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 [26].
- 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 [11], omitting the parts that are irrelevant for our subsequent technical development.

Transactions In its simplest form, a Bitcoin transaction allows a participant 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: $(\mathsf{versig}(pk_{\mathrm{A}}, \mathsf{rtx.wit}), 1₿)$

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: $(\mathsf{versig}(pk_{\mathsf{B}}, \mathsf{rtx.wit}), 1฿)$

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 [11], we represent transactions as tuples of the form $(\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock})$, where:

- in is the list of *inputs*. Each input is a pair (T, i), referring to the i-th output of transaction T.
- wit is the list of *witnesses*, of the same length as the list of inputs. Intuitively, for each input (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 pair (e, v), where the component 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 $(\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock})$, and we denote with $\mathsf{T}.\mathsf{f}$ the content of field f of transaction T. For uniformity, we assume that $\mathsf{absLock}$ is a list of unit length; we omit null values in $\mathsf{absLock}$ and $\mathsf{relLock}$. In transaction fields, we represent a list $\ell_1 \cdots \ell_n$ as $1 \mapsto \ell_1, \ldots, n \mapsto \ell_n$, or just as ℓ_1 when $n = 1$. When clear from the context, we just write the name A of a participant 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 $sig_{sk_{\mathsf{A}}}(\mathsf{T})$.

Scripts Bitcoin scripts are small programs written in a non-Turing equivalent language. Whoever provides a witness that makes the script evaluate to "true",

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

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

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

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

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

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

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

Fig. 1: Semantics of Bitcoin scripts.

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 they evaluate to false. 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 [11] 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 $\mathsf{T}_{i'}.\mathsf{in}(j') = (\mathsf{T}_i, j)$.

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:

$$
\begin{array}{rcll}
e & ::= & \cdots \mid \mathsf{ctx}.\mathsf{f}(e) & \text{access part of the current transaction} \\
& & \mid \mathsf{rtx}.\mathsf{f}(e) & \text{access part of the redeeming transaction} \\
& & \mid \mathsf{outidx} & \text{index of the redeemed output} \\
& & \mid \mathsf{inidx} & \text{index of the redeeming input} \\
& & \mid \mathsf{verscr}(e,e) & \text{covenant} \\
& & \mid \mathsf{verrec}(e) & \text{recursive covenant}
\end{array}
$$

Fig. 2: Extended Bitcoin scripts.

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' is not spent in $\mathbf{B}$;
   (c) $[\![\mathsf{T}'.\mathsf{out}(j)]\!]_{\mathsf{T}_n,i} = \mathit{true}$;
2. $n \geq \mathsf{T}_n.\mathsf{absLock}$;
3. for each input i of T_n, if $\mathsf{T}_n.\mathsf{in}(i) = (\mathsf{T}_h, j)$ then $n - h \geq \mathsf{T}_n.\mathsf{relLock}(i)$;
4. the sum of the amounts of the inputs of T 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 transaction T_i in the blockchain 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 an element to the outputs, making them triples of the form $(\boldsymbol{a}, e, v)$, where $\boldsymbol{a}$ is a sequence of values. Intuitively, the extra element can be used to encode a state within transactions.
- we add script operators to access *any* part of the current transaction and of the redeeming one (by contrast, in pure Bitcoin a script 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 start by introducing some notation to access the parts of transaction outputs. Namely, we extend the range of the meta-variable f to include arg, scr, and val, with the following meaning. If $\mathsf{T}.\mathsf{out}(i) = (\boldsymbol{a}, e, v)$, then $\mathsf{T}.\mathsf{arg}(i) = \boldsymbol{a}$, $\mathsf{T}.\mathsf{scr}(i) = e$, and $\mathsf{T}.\mathsf{val}(i) = v$.

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.

$$[\![\mathsf{ctx}.\mathsf{f}(e)]\!]_{\mathsf{T},i} = \mathsf{T}'.\mathsf{f}([\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{rtx}.\mathsf{f}(e)]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{f}([\![e]\!]_{\mathsf{T},i}) \text{ if } \mathsf{f} \neq \mathsf{wit} \text{ or } [\![e]\!]_{\mathsf{T},i} = i$$

$$[\![\mathsf{outidx}]\!]_{\mathsf{T},i} = j \qquad [\![\mathsf{inidx}]\!]_{\mathsf{T},i} = i$$

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

Fig. 3: Semantics of extended scripts, where $(\mathsf{T}', j) = \mathsf{T}.\mathsf{in}(i)$.

From them, it is possible to infer the current transaction $\mathsf{T}' = \mathit{fst}(\mathsf{T}.\mathsf{in}(i))$, and the index $j = \mathit{snd}(\mathsf{T}.\mathsf{in}(i))$ of the redeemed output. The operator $\mathsf{ctx}.\mathsf{f}(k)$ evaluates to the part $\mathsf{f}(k)$ of the current transaction; similarly, $\mathsf{rtx}.\mathsf{f}(k)$ 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. For coherence with pure Bitcoin, we make the semantics of $\mathsf{rtx}.\mathsf{wit}(k)$ defined only if k is the index of the redeeming input. We use $\mathsf{rtx}.\mathsf{wit}$ as syntactic sugar for $\mathsf{rtx}.\mathsf{wit}(\mathsf{inidx})$. 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 covenants through a series of use cases. At the best of our knowledge, we believe these use cases cannot be obtained in pure Bitcoin. Below, we denote with $\mathsf{U}_{\mathsf{A}}^{v\text{฿}}$ a transaction with a single unspent output of the form $(\varepsilon, \mathsf{versig}(\mathsf{A}, \mathsf{rtx}.\mathsf{wit}), v\text{฿})$.

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฿ 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฿ have been gathered.

To fund the campaign, a contributor A_i publishes the transaction T_i in Figure 4 (left). Its output script is a disjunction between two conditions. The first condition requires that the output at index 1 of the redeeming transaction pays at least v฿ to Z. The second condition allows A_i to redeem her contribution after the expiration date t.

When contributors have deposited enough funds (i.e., there are n transactions $\mathsf{T}_1, \ldots, \mathsf{T}_n$ with $v' = v_1 + \cdots v_n \geq v$), the transaction T_{Z} can be appended by Z to the blockchain. After that, Z can redeem v'฿.

T_i	T_Z
in: $\mathsf{U}^{v_i\text{฿}}_{\mathsf{A}_i}$	in: $1 \mapsto (\mathsf{T}_1, 1), \ldots, n \mapsto (\mathsf{T}_n, 1)$
wit: $\cdots$	wit: $1 \mapsto sig_{\mathsf{Z}}(\mathsf{T}_Z), \ldots, n \mapsto sig_{\mathsf{Z}}(\mathsf{T}_Z)$
out: $(\varepsilon, (\mathsf{versig}(\mathsf{Z}, \mathsf{rtx.wit})$ and $\mathsf{rtx.val}(1) \geq v)$ or $\mathsf{absAfter}\ t : \mathsf{versig}(\mathsf{A}_i, \mathsf{rtx.wit}), v_i\text{฿})$	out: $(\varepsilon, \mathsf{versig}(\mathsf{Z}, \mathsf{rtx.wit}), v'\text{฿})$

Fig. 4: Transactions for the crowdfunding contract.

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. We achieve this by constraining the val part of the redeeming transaction, without using covenants.

4.2 Non-fungible tokens

A non-fungible token represents the ownership of a physical or logical asset. 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. Consequently, the only operation they support is the transfer between users, whereas fungible tokens support split and merge operations.

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

$$e_{NFT} \;=\; \mathsf{versig}(\mathsf{ctx.arg}(1), \mathsf{rtx.wit})\ \mathsf{and}\ \mathsf{verrec}(1)\ \mathsf{and}\ \mathsf{rtx.val}(1) = 1$$

User A creates a new token by depositing 1฿ in T_0, where she sets $\mathsf{T}_0.\mathsf{arg}(1)$ to her public key, to declare her ownership over the token. When A wants to transfer the token to B, she creates another transaction, like T_1 in Figure 5, setting $\mathsf{T}_1.\mathsf{arg}(1)$ to B's public key.

To spend T_0, the transaction T_1 must satisfy the following conditions: (i) the wit field contains the signature of the current owner; (ii) the script at index 1 is equal to that at the same index in T_0; (iii) the output at index 1 has 1฿ value, to preserve the integrity of the token. At this point, B can transfer the token to another user, by appending a transaction which redeems T_1.

The design flaw, already spotted in [26], is exploited by the transaction T_2 in Figure 5. Suppose we have two unspent transactions, T_1 and T'_1, both representing a token. The transaction T_2 can spend both of them, since it complies with all the validity conditions: indeed, the script e_{NFT} only constrains the first output of the redeeming transaction, while the other outputs are only subject to the validity conditions (in particular, that the sum of their values does not exceed the value in input). Actually, T_2 destroys one of the tokens, and removes the covenant from the other one.

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

$$e'_{NFT} \;=\; \mathsf{versig}(\mathsf{ctx.arg}(\mathsf{inidx}), \mathsf{rtx.wit})\ \mathsf{and}\ \mathsf{verrec}(\mathsf{inidx})\ \mathsf{and}\ \mathsf{rtx.val}(\mathsf{inidx}) = 1$$

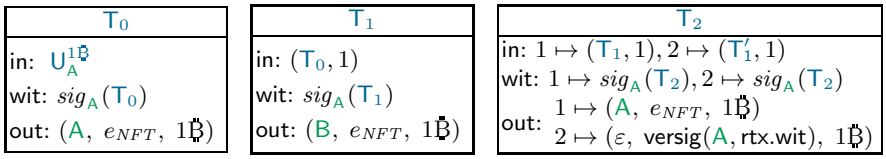

Fig. 5: Transactions of the flawed non-fungible token. The transaction T_2 exploits the flaw to destroy a token, and transfer its value to A.

The new script requires that the index of the input redeeming T_0 is equal to the index of the output "propagating" the token. After this change, a transaction redeeming two tokens must produce two tokens (so, T_2 would not be valid).

An alternative approach to solve the issue, originally proposed in [26], is to add a unique identifier id to each token, e.g. by amending the script as follows:

$$e''_{NFT} \ = \ e_{NFT} \ \mathsf{and} \ id = id$$

This makes two tokens distinguishable. For instance, if the tokens in T_1 and T'_1 are distinguishable, T_2 cannot redeem both of them.

4.3 Vaults

In Bitcoin, transaction outputs are usually secured by cryptographic keys. Whoever knows a key corresponding to transactions can redeem it: in case of key theft, the legitimate owner cannot employ any defence mechanism. Vault transactions, introduced in [26], are a technique to mitigate this issue, by allowing the legitimate owner to abort the transfer.

We implement a basic version of the vault through the following script:

$$e_V \ = \ \begin{array}{l} \mathsf{versig}(\mathsf{A}, \mathsf{rtx.wit}) \ \mathsf{and} \\ \mathsf{verscr}(1, \mathsf{relAfter}\ 100 : \mathsf{versig}(\mathsf{ctx.arg}(\mathsf{inidx}), \mathsf{rtx.wit})\ \mathsf{or}\ \mathsf{versig}(\mathsf{Ar}, \mathsf{rtx.wit})) \end{array}$$

To create a vault, A deposits $1฿$ in T_V in Figure 6. The transaction can be redeemed with the signature of A, but only by a de-vaulting transaction such as T_S, with the script:

$$e_S \ = \ \mathsf{relAfter}\ 100 : \mathsf{versig}(\mathsf{ctx.arg}(\mathsf{inidx}), \mathsf{rtx.wit})\ \mathsf{or}\ \mathsf{versig}(\mathsf{Ar}, \mathsf{rtx.wit})$$

The output of the de-vaulting transaction T_S can be spent by the participant set in its argument, but only after a certain time. Before the time expires, A can cancel the transfer by spending T_S with her recovery key Ar.

A recursive vault The vault in Figure 6 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 key theft.

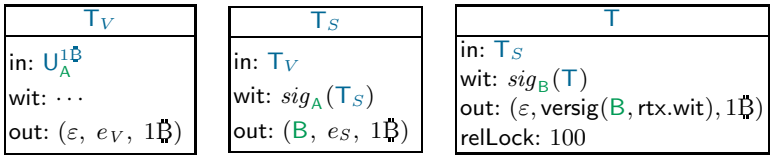

Fig. 6: Transactions for the basic vault.

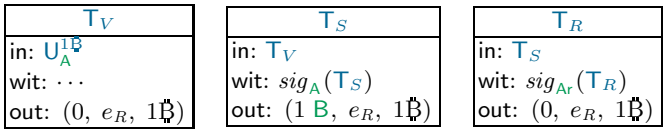

Fig. 7: Transactions for the recursive vault.

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

$$e_R \ = \ \begin{array}{l} \mathsf{if}(\mathsf{ctx.arg}(1).1 = 0) \\ \mathsf{then\ versig}(\mathsf{A}, \mathsf{rtx.wit})\ \mathsf{and\ verrec}(1)\ \mathsf{and\ rtx.arg}(1).1 = 1 \\ \mathsf{else}\ \ (\mathsf{relAfter}\ 100 : \mathsf{versig}(\mathsf{ctx.arg}(1).2, \mathsf{rtx.wit}) \\ \qquad\quad \mathsf{or\ versig}(\mathsf{Ar}, \mathsf{rtx.wit})\ \mathsf{and\ verrec}(1)\ \mathsf{and\ rtx.arg}(1).1 = 0) \end{array}$$

In this version, the vault and de-vaulting transactions (in Figure 7) have the same script. The first argument of the script encodes the contract state: 1 models the vault state, and 2 to the de-vaulting state. The recovery key Ar can only be used to append the re-vaulting transaction T_R, locking again the bitcoin into the vault. Key theft is now 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.

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. We design a pyramid scheme in Bitcoin using the transactions in Figure 8, where:

$$e_P \ = \ \begin{array}{l} \mathsf{verscr}(1, \mathsf{versig}(\mathsf{ctx.arg}(\mathsf{outidx}), \mathsf{rtx.wit})) \\ \mathsf{and\ rtx.arg}(1) = \mathsf{ctx.arg}(\mathsf{outidx})\ \mathsf{and\ rtx.val}(1) = 2 \\ \mathsf{and\ verrec}(2)\ \mathsf{and\ verrec}(3) \end{array}$$

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 e_P ensures that this redeeming transaction has the form of T_1 in Figure 8, i.e. $\mathsf{out}(1)$ transfers 2฿ to A_0, while the scripts in

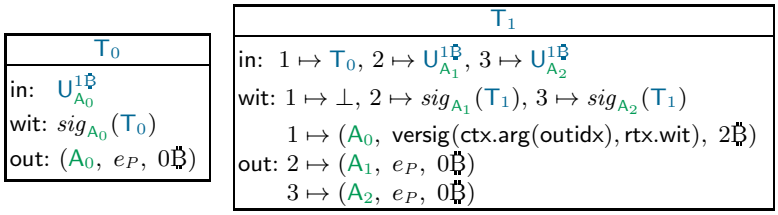

Fig. 8: Transactions for the pyramid scheme.

$\mathsf{out}(2)$ 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 toll v_0 to become the "king". Another user can usurp the throne by paying $v_1 = 1.5v_0$ to the old king, and so on until new usurpers are available. Of course this leads to an exponential growth of the currency 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. Although the logic to distribute currency 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 9 for the first two kings, A_0 and A_1, where:

$$e_{KET} = \begin{array}{l} \mathsf{verrec}(1)\ \text{ and }\ \mathsf{rtx.arg}(2) = \mathsf{ctx.arg}(1)\ \text{ and }\ \mathsf{rtx.val}(2) \geq 2\,\mathsf{ctx.val}(2)\ \text{ and } \\ \mathsf{verscr}(2, \mathsf{versig}(\mathsf{ctx.arg}(2), \mathsf{rtx.wit})) \end{array}$$

The transactions use $\mathsf{arg}(1)$ to store the (public key of) the new king, and $\mathsf{arg}(2)$ for the old one. The clause $\mathsf{rtx.arg}(2) = \mathsf{ctx.arg}(1)$ in e_{KET} ensures that the old king is correctly recorded in the redeeming transaction, while $\mathsf{rtx.val}(2) \geq 2\,\mathsf{ctx.val}(2)$ ensures that his compensation is twice the value he paid. Finally, the verscr guarantees that the old king can redeem his compensation via $\mathsf{out}(2)$.

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

T_0	
in:	$\mathsf{U}^{v_0\text{฿}}_{\mathsf{A}_0}$
wit:	$sig_{\mathsf{A}_0}(\mathsf{T}_0)$
out:	$1 \mapsto (\mathsf{A}_0,\ e_{KET},\ 0\text{฿})$ $2 \mapsto (\mathsf{A}_0,\ \mathsf{versig}(\mathsf{A}_0, \mathsf{rtx.wit}),\ v_0\text{฿})$

T_1	
in:	$1 \mapsto (\mathsf{T}_0, 1),\ 2 \mapsto \mathsf{U}^{v_1\text{฿}}_{\mathsf{A}_1}$
wit:	$1 \mapsto \bot,\ 2 \mapsto sig_{\mathsf{A}_1}(\mathsf{T}_1)$
out:	$1 \mapsto (\mathsf{A}_1,\ e_{KET},\ 0\text{฿})$ $2 \mapsto (\mathsf{A}_0,\ \mathsf{versig}(\mathsf{ctx.arg}(2), \mathsf{rtx.wit}),\ v_1\text{฿})$

Fig. 9: Transactions for King of the Ether Throne.

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 transactions, and modifying the entries of the UTXO set to link each unspent output to the enclosing transaction. Note that the witnesses of this transaction must be left available, in order to implement $\mathsf{ctx.wit}(e)$.

The language primitives that check the redeeming transaction script, verscr and verrec, can be implemented through an opcode similar to CheckOutputVerify described in [26]. While [26] 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., $\mathsf{versig}(\mathsf{ctx.arg}(1), \mathsf{rtx.wit})$. Thus, to check if two scripts are the same we just need to compare their hashes, while [26] needs to instantiate the placeholders. Similarly, we can use the hash of the script within verscr. The work [27] 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 [5,11], 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.

Note that, even though the use cases in Section 4 make extensive use of non-standard scripts, they can be encoded into standard transaction using P2SH, as done in [5,11]. Crucially, the hash also covers the arg part of the output, 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 [15] 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 [13] 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 [10], BitML has known expressiveness limits, given by the requirement to have pure Bitcoin as its compilation target. For instance, it is not possible to specify in BitML (and, we believe, not even in pure Bitcoin) recursive contracts, unless all participants agree to perform the recursive call [13]. 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.

A possible extension of BitML is the construct:

$$?\boldsymbol{x}\ \texttt{if}\, b.\ \mathtt{X}\langle \boldsymbol{x} \rangle$$

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 $\mathtt{X}\langle \boldsymbol{x} \rangle$, which will reduce according to the equation defining `X`.

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

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

The contract $\mathtt{X}\langle \mathsf{p}, v \rangle$ models a state where p 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 greater than $2v$. This participant can choose the value for q, i.e. the new king. The contract proceeds as $\mathtt{Y}\langle \mathsf{p}, \mathsf{q}, \mathsf{val} \rangle$, which has two parallel branches. The first branch makes val ฿ available to the old king; the second branch has zero value, and it reboots the game, recording the new king q and his investment.

The output of the compiler is the script in Figure 10. Executing each step of the BitML contract corresponds, in Bitcoin, to appending a transaction containing the script in the first output. The script implements a state machine, using $\mathsf{arg}.1$ to record the current state. The other arguments of $\mathsf{out}(1)$ store, respectively, the old king, the new king, and the value v. The $\mathsf{verrec}(1)$ at line 1 ensures that the script is preserved in $\mathsf{out}(1)$. To pay the old king, we use the verscr at line 17, which constrains the script in $\mathsf{out}(2)$ of the transaction which corresponds to the BitML state $v \to \texttt{withdraw}\ \mathsf{p} \mid 0 \to \mathtt{X}\langle \mathsf{q}, v \rangle$.

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฿ as a bet. The game first makes each player commit to a secret, using a timed-commitment protocol [18].

```
verrec(1) and
  if ctx.arg.1 = 0 then                  // state X(A,1)
        rtx.arg.1 = 1                    // advance the state
    and rtx.val(1) > 2 * ctx.val(1)      // investment at least 2v
    and rtx.arg.2 = ctx.arg.2            // preserve old king (p)
    and rtx.arg.4 = 2 * ctx.val(1)       // instantiate v
  else if ctx.arg.1 = 1 then             // state Y(p,q,v)
        rtx.arg.1 = 2                    // advance the state
    and rtx.val(1) = ctx.val(1)          // preserve value in out(1)
    and rtx.arg.2 = ctx.arg.2            // preserve old king (p)
    and rtx.arg.3 = ctx.arg.3            // preserve new king (q)
    and rtx.arg.4 = ctx.arg.4            // preserve v
  else if ctx.arg.1 = 2 then             // state v->withdraw p | 0->X<q,v>
        rtx.arg.1 = 3                    // advance the state
    and rtx.arg.2 = ctx.arg.3            // new king in ctx = old king in rtx
    and rtx.arg.4 = ctx.arg.4            // preserve v
    and verscr(2, versig(ctx.arg(2).1, rtx.wit)) //pay the old king
    and rtx.arg(2).1 = ctx.arg(1).2      // set arg(2) to the old king
    and rtx.val(2) = ctx.val(1)          // set out(2) value to v
  else if ctx.arg.1 = 3 then             // state X<q,v>
        rtx.arg.1 = 1                    // advance the state
    and rtx.val(1) > 2 * ctx.arg.4       // investment at least 2v
    and rtx.arg.2 = ctx.arg.2            // preserve old king (p)
    and rtx.arg.4 = 2 * ctx.arg.4        // update v
```

Fig. 10: Script for King of the Ether Throne, obtained by compiling BitML.

The secrets are then revealed, and the winner of a flip is determined as a function of the two secrets. The game starts another coin 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 $\mathtt{X}_{\mathsf{A}}\langle \mathsf{C} \rangle$, where $\mathsf{C} \neq \mathsf{A}, \mathsf{B}$, using the following defining equations:

$$\mathtt{X}_{\mathsf{A}}\langle \mathsf{w} \rangle = \mathsf{A} : ?h_{\mathsf{A}} \,.\, \mathtt{X}_{\mathsf{B}}\langle \mathsf{w}, h_{\mathsf{A}} \rangle \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B}$$

$$\mathtt{X}_{\mathsf{B}}\langle \mathsf{w}, h_{\mathsf{A}} \rangle = \mathsf{B} : ?h_{\mathsf{B}} \,.\, \mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A}$$

$$\mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle = ?s_{\mathsf{A}}\ \mathtt{if}\ H(s_{\mathsf{A}}) = h_{\mathsf{A}} \,.\, \mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B}$$

$$\mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle = ?s_{\mathsf{B}}\ \mathtt{if}\ H(s_{\mathsf{B}}) = h_{\mathsf{B}}\ \mathsf{and}\ 0 \leq s_{\mathsf{B}} \leq 1 \,.\, \mathtt{W}\langle \mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}} \rangle + \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A}$$

$$\begin{aligned} \mathtt{W}\langle \mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}} \rangle = {} & \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} = \mathsf{A} : \mathtt{withdraw}\ \mathsf{A} \\ & + \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} \neq \mathsf{A} : \mathtt{X}_{\mathsf{A}}\langle \mathsf{A} \rangle \\ & + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} = \mathsf{B} : \mathtt{withdraw}\ \mathsf{B} \\ & + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} \neq \mathsf{B} : \mathtt{X}_{\mathsf{A}}\langle \mathsf{B} \rangle \end{aligned}$$

The contract $\mathtt{X}_{\mathsf{A}}\langle \mathsf{w} \rangle$ 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, $\mathtt{X}_{\mathsf{B}}\langle \mathsf{w} \rangle$ models B's turn to commit. In $\mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle$, A must reveal her secret s_{A}, or otherwise lose her deposit. The contract $\mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle$ 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 [15]). 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 seem to be unfeasible in current 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. 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 [10]. In particular, note that the size of a script grows with the number of "states" of a contract (see e.g. Figure 10), and so complex contracts would easily violate the 520 bytes limit. The introduction of Taproot [28] would be helpful to mitigate this limitation. 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 limitation to the expressiveness of contracts derives from the fact that our scripts can only access the current and redeeming transactions: e.g., it is not possible to express conditions on the value of a transaction in the inputs of the current one.

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 [7], 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 [15], 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 [16]. Note that our extension of the UTXO model is more restrictive than the one in [20], as the latter abstracts from the script language, just assuming that scripts denote any pure functions [32]. This added flexibility can be exploited to design expressive high-level contract languages like Marlowe [31] and Plutus [19].

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.kingoftheet
4. Understanding the DAO attack (June 2016), http://www.coindesk.com/understanding-dao-hack-journalists/
5. Balzac: Bitcoin abstract language, analyzer and compiler. https://blockchain.unica.it/balzac/ (2018)
6. 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
7. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Modeling Bitcoin contracts by timed automata. In: International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 8711, pp. 7–22. Springer (2014). https://doi.org/10.1007/978-3-319-10512-3_2
8. 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, first appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784
9. 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
10. 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
11. 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
12. 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
13. Bartoletti, M., Murgia, M., Zunino, R.: Renegotiation and recursion in Bitcoin contracts. In: COORDINATION. LNCS, Springer (2020). https://doi.org/10.1007/978-3-662-58387-6, (To appear)
14. 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

15. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. In: ACM CCS (2018). https://doi.org/10.1145/3243734.3243795
16. Bartoletti, M., Zunino, R.: Verifying liquidity of Bitcoin contracts. In: POST. LNCS, vol. 11426, pp. 222–247. Springer (2019)
17. 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
18. 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
19. Brünjes, L., Gabbay, M.J.: UTxO- vs account-based smart contract blockchain programming paradigms. CoRR **abs/2003.14271** (2020)
20. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: Workshop on Trusted Smart Contracts (2020)
21. Dashjr, L.: BIP 0002 (2016), https://en.bitcoin.it/wiki/BIP_0002
22. 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
23. 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
24. Maxwell, G.: CoinCovenants using SCIP signatures, an amusingly bad idea (2013), https://bitcointalk.org/index.php?topic=278122.0
25. 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
26. 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
27. 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
28. 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
29. Poon, J., Dryja, T.: The Bitcoin Lightning Network: Scalable off-chain instant payments (2015), https://lightning.network/lightning-network-paper.pdf
30. Rubin, J.: CHECKTEMPLATEVERIFY (2020), BIP 119, https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki
31. 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
32. Zahnentferner, J.: An abstract model of utxo-based cryptocurrencies with scripts. Cryptology ePrint Archive, Report 2018/469 (2018), https://eprint.iacr.org/2018/469