A formal model of Bitcoin transactions

Nicola Atzei[1], 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. We propose a formal model of Bitcoin transactions, which is sufficiently abstract to enable formal reasoning, and at the same time is concrete enough to serve as an alternative documentation to Bitcoin. We use our model to formally prove some well-formedness properties of the Bitcoin blockchain, for instance that each transaction can only be spent once. We release an open-source tool through which programmers can write transactions in our abstract model, and compile them into standard Bitcoin transactions.

1 Introduction

In recent years we have observed a growing interest around *cryptocurrencies*. Bitcoin [12], the first decentralized cryptocurrency, was introduced in 2009, and through the years it has consolidated its position as the most popular one. Bitcoin and other cryptocurrencies have pushed forward the concept of decentralization, providing means for reliable interactions between mutually distrusting parties on an open network.

The nodes of the Bitcoin network maintain a public and immutable data structure, called *blockchain*. The blockchain stores the historical record of all transfers of bitcoins, which are referred to as *transactions*. When a node updates the blockchain, the other nodes verify if the appended transactions are valid, e.g. by checking if the conditions specified in *scripts* are satisfied. Scripts are programmable boolean functions: in their standard (and mostly used) form they verify a digital signature against a public key. Since the blockchain is immutable, tampering with a stored transaction would result in the invalidation of all the subsequent ones. Updating the state of the blockchain, i.e. appending new transactions, requires solving a moderately difficult cryptographic puzzle. In case of conflicting updates, the chain that required the largest computational effort is considered the valid one. Hence, the immutability and the consistency of the blockchain is bounded by the total computational power of honest nodes. An adversary with enough resources can append invalid transactions, e.g. with incorrect digital signatures, or rewrite a part of the blockchain, e.g. to perform a *double-spending attack*. The attack consists in paying someone by publishing a transaction on the blockchain, and then removing it (making the funds unspent).

Besides the intended monetary application, the Bitcoin blockchain can be seen as a way to consistently maintain the state of a system over a peer-to-peer network, without the need of a trusted authority. If the system is a currency, its

state is the amount of funds in each account. This concept can be generalised to the case where the system is a *smart contract* [14], namely an executable computer protocol which can also handle transfers of currency. The idea of exploiting the Bitcoin blockchain to build smart contracts has recently been explored by several works. Lotteries [2,4,5,10], gambling games [9], contingent payments [3], covenants [11,13], and other kinds of fair computations [1,8] are some examples of the capabilities of Bitcoin as a platform for smart contracts.

Smart contracts often rely on features of Bitcoin that go beyond the standard transfers of currency. For instance, while the vast majority of Bitcoin transactions uses scripts only to verify signatures, smart contracts like the above-mentioned ones exploit more complex scripts, e.g. to determine the winner of a lottery, or to check if a secret has been revealed. Smart contracts may also exploit other (infrequently used) features of Bitcoin, e.g. various signature modifiers, and temporal constraints on transactions.

As a matter of fact, using these advanced features to design a new smart contract is not a trivial matter, for two reasons. First, while the overall behaviour of Bitcoin is clear, the details of many of its crucial aspects are poorly documented. To understand the details of how a mechanism actually works, one has to explore various web pages (often inaccurate, or inconsistent, or overly technical), and eventually resort to the source code of the Bitcoin client[3] to have the correct answer. Second, the description of advanced features is often too concrete to be effectively used in the design and analysis of a smart contract (indeed, in many cases the only available description coincides with the implementation).

Contributions. We propose a formal model of Bitcoin transactions. This model is abstract enough to allow for formal reasoning on the behaviour of Bitcoin transactions. For instance, we use our model to formally prove some properties of the Bitcoin blockchain, e.g. that transactions cannot be spent twice (Theorem 1), and that the overall value contained in the blockchains (excluding the coinbase transactions) is decreasing (Theorem 2).

Our model formally specifies some poorly documented features of Bitcoin, e.g. transaction signatures and signature modifiers (Definition 4), output scripts (Definitions 1 and 7), multi-signature verification (Definition 6), Segregated Witnesses (Definitions 2 and 9), paving the way towards automatic verification.

We make available an open-source tool[4] which translates transactions specified in our model to *standard* Bitcoin transactions.

Structure of the paper. Section 2 briefly recaps Bitcoin transactions, which we formalise in Section 3. Besides transactions, we also provide an high-level model of the blockchain, and we study its basic properties. In Section 4 we illustrate, through a basic case study, the impact of the Segregated Witness feature on the expressiveness of Bitcoin smart contracts. In Section 5 we show how to translate transactions from our model to standard Bitcoin transactions. We discuss the differences between our model and the actual Bitcoin in Section 6.

[3] https://github.com/bitcoin/bitcoin.

[4] https://github.com/bitcoin-transaction-model/bitcoin-transaction-model.

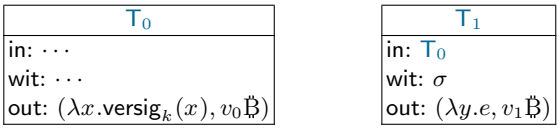

Fig. 1: Two Bitcoin transactions.

2 Bitcoin transactions in a nutshell

We now give a minimalistic introduction to the behaviour of Bitcoin transactions (see [6] for a general survey on the other aspects of Bitcoin).

Users interact with Bitcoin through *addresses*, which they can freely generate. *Transactions* describe transfers of bitcoins ($\ddot{\mathrm{B}}$) between addresses. The log of all transactions is recorded on a public, immutable and decentralised data structure called *blockchain*. To explain how the blockchain works, consider the transactions T_0 and T_1 displayed in Figure 1. The transaction T_0 contains $v_0\ddot{\mathrm{B}}$, which can be *redeemed* by putting on the blockchain a transaction (e.g., T_1), whose in field is a reference to T_0. To redeem T_0, the *witness* of the redeeming transaction (the value in its wit field) must make the *output script* of T_0 (the first element of the pair in the out field) evaluate to true. When this happens, the value of T_0 is transferred to the new transaction, and T_0 is no longer redeemable.

In the example displayed before, the output script of T_0 evaluates to true when receiving a digital signature on the redeeming transaction T_1, with a given key pair k. We denote with $\mathsf{versig}_k(x)$ the verification of the signature x on the redeeming transaction: of course, since the signature must be included in the witness of the redeeming transaction, it will consider all the parts of that transaction *except* its wit field. We assume that σ is the signature of T_1.

Now, assume that the blockchain contains T_0, not yet redeemed, and someone tries to append T_1. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the output script of T_0, by instantiating its formal parameter x to the signature σ in the witness of T_1. The function $\mathsf{versig}_k(\sigma)$ verifies that σ is actually the signature of T_1: therefore, the output script succeeds, and T_1 redeems T_0. Subsequently, a new transaction can redeem T_1 by satisfying its output script $\lambda y.e$ (not specified in the figure).

Bitcoin transactions may be more general than the ones illustrated by the previous example. First, there can be multiple inputs and outputs. Each output has an associated output script and value, and can be redeemed independently from the others. Consequently, in fields must specify which output they are redeeming. A transaction with multiple inputs associates a witness to each of them. The sum of the values of all the inputs must be greater or equal to the sum of the values of all the outputs, otherwise the transaction is considered invalid. In its general form, the output script is a program in a (non Turing-complete) scripting language, featuring a limited set of logic, arithmetic, and cryptographic operators. Finally, a transaction can specify time constraints (absolute, or relative to its input transactions) about when it can appear on the blockchain.

$\mathsf{A}, \mathsf{B}, \ldots \in \mathsf{Part}$	Participants	$e, e', \ldots \in \mathsf{Exp}$	Script expressions
$x, y, \ldots \in \mathsf{Var}$	Variables	$\mathsf{T}, \mathsf{T}', \ldots \in \mathsf{Tx}$	Transactions
$\nu, \nu', \ldots \in \mathsf{Den}$	Denotations, i.e.:	μ, μ'	Signature modifier
$k, k' \ldots \in \mathbb{Z}$	Constants	$\mathsf{sig}_k^{\mu,i}(\mathsf{T})$	Transaction signature
$t, t' \ldots \in \mathbb{N}$	Time	$\mathsf{ver}_k(\sigma, \mathsf{T}, i)$	Signature verification
$v, v' \ldots \in \mathbb{N}$	Currency values	$\mathsf{T}, i \models \lambda \boldsymbol{x}.e$	Script verification
$\sigma, \sigma', \ldots \in \mathbb{Z}$	Signatures	$(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$	Transaction redeem
$\mathit{true}, \mathit{false}$	Boolean values	$\mathbf{B} = (\mathsf{T}_1, t_1) \cdots$	Blockchains
$\bot$	Undefined	$\mathbf{B} \rhd (\mathsf{T}, t)$	Consistent update

Table 1: Summary of notation.

3 A formal model of Bitcoin transactions

In this section we introduce a formal model of Bitcoin transactions. We start in Section 3.1 by defining the scripts that can be used in transaction outputs. Then, in Section 3.2 we formalise transactions, and in Section 3.3 we define a signature scheme for them. Sections 3.4 and 3.5 give semantics, respectively, to scripts and transactions. In Section 3.6 we model the Bitcoin blockchain, and in particular we define the crucial notion of *consistency*, which corresponds to the one enforced by the Bitcoin consensus protocol. We then state a few results about consistent blockchains (their proofs are in Appendix A).

We start by introducing some auxiliary notation. We assume several sets, ranged over by meta-variables as shown in the left column of Table 1. We use the bold notation to denote finite sequences of elements. We denote with $\boldsymbol{x}_i$ the i-th element of a sequence $\boldsymbol{x}$, i.e. $\boldsymbol{x}_i = x_i$ if $\boldsymbol{x} = x_1 \ldots x_n$, and with $\boldsymbol{x}_{i..j}$ the subsequence of $\boldsymbol{x}$ starting from the i-th element and ending to the j-th element. We denote with $|\boldsymbol{x}|$ the number of elements of $\boldsymbol{x}$, and with $[]$ the empty sequence. We denote with $f : A \rightharpoonup B$ a *partial* function f from A to B, with $\mathrm{dom}\, f$ the *domain* of f, i.e. the subset of A where f is defined, and with $\mathrm{ran}\, f$ the *range* of f, i.e. $\mathrm{ran}\, f = \{f(x) \,|\, x \in \mathrm{dom}\, f\}$. We use $\bot$ to represent an "undefined" element; in particular, when the element is a partial function, $\bot$ denotes the function with empty domain. For a pair (x, y), we define $\mathit{fst}(x, y) = x$ and $\mathit{snd}(x, y) = y$.

3.1 Scripts

Each output in a Bitcoin transaction contains a script, which is used to establish when the output can be redeemed by another transaction. Intuitively, a script is a first-order function (written in a non Turing-equivalent language), which is applied to the witness provided by the redeeming transaction. The output can be redeemed only if such function application evaluates to true.

In our model, we abstract from the actual stack-based scripting language implemented in Bitcoin[5], by using instead a minimalistic language of expressions.

[5] https://en.bitcoin.it/wiki/Script.

Definition 1 (Scripts). *We define the set* Exp *of* script expressions *(ranged over by* $e, e', \dots$*) as follows:*

$$e ::= x \mid k \mid e + e \mid e - e \mid e = e \mid e < e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid |e| \mid \mathsf{H}(e) \mid \mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{e}) \mid \mathsf{absAfter}\ t : e \mid \mathsf{relAfter}\ t : e$$

We denote with Script *the set of terms of the form* $\lambda \boldsymbol{z}.e$ *such that all the variables in* e *occur in* $\boldsymbol{z}$*.*

Besides some basic arithmetic and logical operators, script expressions include a few operators inspired from the actual Bitcoin scripting language. The expression $|e|$ denotes the size, in bytes, of the evaluation of e. The expression $\mathsf{H}(e)$ evaluates to the hash of e. The expression $\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{e})$ takes as arguments a sequence of m script expressions, representing signatures of the enclosing transactions, and a sequence of n public keys. Intuitively, it evaluates to true whenever the provided signatures are verified by using m out of the n provided keys. The expressions $\mathsf{absAfter}\ t : e$ and $\mathsf{relAfter}\ t : e$ define temporal constraints (see Section 3.4). They evaluate as e if the constraints are satisfied, otherwise they fail.

Notation 1. We use the following syntactic sugar for expressions: (i) *false* to denote $1 = 0$ (ii) *true* to denote $1 = 1$ (iii) $e \wedge e'$ to denote $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ \mathit{false}$ (iv) $e \vee e'$ to denote $\mathsf{if}\ e\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ e'$ (v) $\mathsf{not}\ e$ to denote $\mathsf{if}\ e\ \mathsf{then}\ \mathit{false}\ \mathsf{else}\ \mathit{true}$.

3.2 Transactions

The following definition formalises Bitcoin transactions.

Definition 2 (Transactions). *We inductively define the set* Tx *of transactions as follows. A transaction* T *is a tuple* $(\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock})$*, where:*

- $\mathsf{in} : \mathbb{N} \rightharpoonup \mathsf{Tx} \times \mathbb{N}$
- $\mathsf{wit} : \mathbb{N} \rightharpoonup \mathbb{Z}^*$*, where* $\operatorname{dom} \mathsf{wit} = \operatorname{dom} \mathsf{in}$
- $\mathsf{out} : \mathbb{N} \rightharpoonup \mathsf{Script} \times \mathbb{N}$
- $\mathsf{absLock} : \mathbb{N}$
- $\mathsf{relLock} : \mathbb{N} \rightharpoonup \mathbb{N}$*, where* $\operatorname{dom} \mathsf{relLock} = \operatorname{dom} \mathsf{in}$

where, for all $i, j \in \operatorname{dom} \mathsf{in}$*,* $\mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$ *and* $i \neq j \implies \mathsf{in}(i) \neq \mathsf{in}(j)$*.*
We denote with $\mathsf{T}.\mathsf{f}$ *the value of field* f *of* T*, for* $\mathsf{f} \in \{\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock}\}$*.*
We say that T *is* initial *when* $\mathsf{T}.\mathsf{in} = \mathsf{T}.\mathsf{relLock} = \bot$ *and* $\mathsf{T}.\mathsf{absLock} = 0$*.*

The fields in and out represent, respectively, the inputs and the outputs of a transaction. There is an input for each $i \in \operatorname{dom} \mathsf{in}$, and an output for each $j \in \operatorname{dom} \mathsf{out}$. When $\mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j)$, it means that the i-th input of T wants to redeem the j-th output of T'. The side condition $i \neq j \Rightarrow \mathsf{in}(i) \neq \mathsf{in}(j)$ ensures that inputs are pairwise distinct. The side condition $\mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$ is related to the *Segregated Witness (SegWit)* feature[6], and it requires that the witness of

[6] This feature, specified in the BIP 141 and activated on August 24th 2017, implies that witnesses are not used in the computation of transaction hashes.

the input transaction is left unspecified. The output $\mathsf{T}'.\mathsf{out}(j)$ is a pair $(\lambda \boldsymbol{z}.e, v)$, meaning that v Satoshis ($1\text{Ƀ} = 10^8$ Satoshis) can be redeemed by whoever can provide a witness which satisfies $\lambda \boldsymbol{z}.e$. Such witness is defined by $\mathsf{T}.\mathsf{wit}(i)$. The fields $\mathsf{T}.\mathsf{absLock}$ and $\mathsf{T}.\mathsf{relLock}(i)$ specify a constraint on when T can be put on the blockchain: the first in absolute terms, whereas the second is relative to the transaction in the input $\mathsf{T}.\mathsf{in}(i)$. More specifically, $\mathsf{T}.\mathsf{absLock} = t$ means that T can appear on the blockchain only after time t. If $\mathsf{T}.\mathsf{relLock}(i) = t$, then T can appear only after time t since the transaction in $\mathsf{T}.\mathsf{in}(i)$ appeared.

To improve readability, we use the following conventions: (i) if T has exactly one input, we denote it by $\mathsf{T}.\mathsf{in}$ (omitting the index, which we assume to be 1); We act similarly for $\mathsf{T}.\mathsf{wit}$, $\mathsf{T}.\mathsf{out}$, and $\mathsf{T}.\mathsf{relLock}$; (ii) if $\mathsf{T}.\mathsf{absLock} = 0$, we omit it (similarly for $\mathsf{T}.\mathsf{relLock}$ when it is $\bot$); (iii) we denote with $\mathit{script}(\mathsf{T}.\mathsf{out}(i))$ and $\mathit{val}(\mathsf{T}.\mathsf{out}(i))$, respectively, the first and the second element of the pair $\mathsf{T}.\mathsf{out}(i)$.

3.3 Transaction signatures

We extend to transactions the signing and verification functions of the signature schemes, denoted respectively as $\mathit{sig}_k(\cdot)$ and $\mathit{ver}_k(\cdot,\cdot)$. For simplicity, although we will always use $k = (k_p, k_s)$ for key pairs, we implicitly assume that $\mathit{sig}_k(\cdot)$ only uses the private part k_s, while $\mathit{ver}_k(\cdot,\cdot)$ only uses the public part k_p.

In Bitcoin, transaction signatures never apply to the whole transaction: users can specify which parts of a transaction are signed (with the exception of the wit field, which is never signed). However, not all possible combinations of transaction parts are possible; the legit ones are listed in Definition 4. In order to specify which parts of a transaction are signed, we first introduce the auxiliary notion of *transaction substitution*.

Definition 3 (Transaction substitutions). *A* transaction substitution Σ *is a function from* Tx *to* Tx. *For a transaction field* f, *we denote with* $\{\mathsf{f} \mapsto d\}$ *the substitution which replaces the value of* f *with* d. *For* $\mathsf{f} \neq \mathsf{absLock}$ *and* $i \in \mathbb{N}$, *we denote with* $\{\mathsf{f}(i) \mapsto d\}$ *the substitution which replaces* $\mathsf{f}(i)$ *with* d. *Further, for* $\circ \in \{<, >, \neq\}$, *we denote with* $\{\mathsf{f}(\circ\, i) \mapsto d\}$ *the substitution which replaces* $\mathsf{f}(j)$ *with* d, *for all* $j \circ i \in \mathrm{dom}\,\mathsf{f}$.

Definition 4 (Signature modifiers). *We define signature modifiers* μ_i *(with* $i \in \mathbb{N}$*) in Figure 2. We associate to each modifier a substitution, and we denote with* $\mu_i(\mathsf{T})$ *the result of applying it to the transaction* T.

Each modifier is represented by a pair of symbols, describing, respectively, the subset of inputs and of outputs being signed (a = all, s = single, n = none). In case of s, the parameter i determines the index of the signed input/output.

Definition 5 (Transaction signatures). *We define the transaction signature (under modifier* μ *and index* i*) and verification functions as follows:*

$$\mathsf{sig}_k^{\mu,i}(\mathsf{T}) = (\mathit{sig}_k(\mu_i(\mathsf{T})), \mu) \qquad \mathsf{ver}_k(\sigma, \mathsf{T}, i) = \mathit{ver}_k(w, \mu_i(\mathsf{T})) \;\; \textit{if } \sigma = (w, \mu)$$

Hereafter, we use $\sigma, \sigma', \ldots$ *to range over transaction signatures.*

$$
\begin{aligned}
aa_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit} \mapsto \bot\} \\
an_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit} \mapsto \bot\}\{\mathsf{out} \mapsto \bot\} \\
as_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit} \mapsto \bot\}\{\mathsf{out}(< i) \mapsto (\mathit{false}, 0)\}\{\mathsf{out}(> i) \mapsto \bot\} \\
sa_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit} \mapsto \bot\}\{\mathsf{in}(1) \mapsto \mathsf{T}.\mathsf{in}(i)\}\{\mathsf{in}(\neq 1) \mapsto \bot\} \\
&\quad \{\mathsf{relLock}(1) \mapsto \mathsf{T}.\mathsf{relLock}(i)\}\{\mathsf{relLock}(\neq 1) \mapsto \bot\} \\
sn_i(\mathsf{T}) &= sa_i(an_i(\mathsf{T})) \\
ss_i(\mathsf{T}) &= sa_i(as_i(\mathsf{T}))
\end{aligned}
$$

Fig. 2: Signature modifiers.

Note that a signature $\sigma = (sig_k(\mu_i(\mathsf{T})), \mu)$ does not contain the index i. Consequently, the verification function requires i to be passed as parameter, i.e. we write $\mathsf{ver}_k(\sigma, \mathsf{T}, i)$. The parameter i will be instantiated by the script verification function (see Definition 8).

Notation 2. Note that $\mathsf{sig}_k^{\mu,i}(\mathsf{T})$ can meaningfully appear within $\mathsf{T}.\mathsf{wit}(i)$, since the signature always neglects the wit field of transactions (as all signature modifiers set wit to $\bot$). In this case, as a shorthand we denote it with sig_k^{μ} (so, neglecting the enclosing transaction T), or just sig_k when $\mu = aa$.

We now extend the signature verification $\mathsf{ver}_k(\sigma, \mathsf{T}, i)$ to the case where, instead of providing a single key k and a single signature σ, one has many keys and signatures, i.e. $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$. Intuitively, if $|\boldsymbol{\sigma}| = m$ and $|\boldsymbol{k}| = n$, the function $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$ implements a m-of-n multi-signature scheme, i.e. it evaluates to true if all the m signatures match (some of) the keys in $\boldsymbol{k}$. The actual definition is a bit more complex, to be coherent with the one implemented in Bitcoin.

Definition 6 (Multi-signature verification). *Let $\boldsymbol{k}$ and $\boldsymbol{\sigma}$ be sequences of (public) keys and signatures such that $|\boldsymbol{k}| \geq |\boldsymbol{\sigma}|$, and let $i \in \mathbb{N}$. For all $m, n \in \mathbb{N}$, we define the function:*

$$
\mathsf{ver}_{\boldsymbol{k}}^{n,m}(\boldsymbol{\sigma}, \mathsf{T}, i) \equiv \begin{cases} \mathit{true} & \mathit{if}\ m = 0 \\ \mathit{false} & \mathit{if}\ m \neq 0\ \mathit{and}\ n = 0 \\ \mathsf{ver}_{\boldsymbol{k}}^{n-1,m-1}(\boldsymbol{\sigma}, \mathsf{T}, i) & \mathit{if}\ m, n \neq 0\ \mathit{and}\ \mathsf{ver}_{\boldsymbol{k}_n}(\boldsymbol{\sigma}_m, \mathsf{T}, i) \\ \mathsf{ver}_{\boldsymbol{k}}^{n-1,m}(\boldsymbol{\sigma}, \mathsf{T}, i) & \mathit{otherwise} \end{cases}
$$

Then, we define $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i) = \mathsf{ver}_{\boldsymbol{k}}^{|\boldsymbol{k}|,|\boldsymbol{\sigma}|}(\boldsymbol{\sigma}, \mathsf{T}, i)$.

Our formalisation of multi-signature verification (Definition 6) follows closely the implementation of Bitcoin, whose stack-based scripting language imposes that the sequence $\boldsymbol{\sigma}$ is read in reverse order. Accordingly, the function ver tries to verify the last signature in $\boldsymbol{\sigma}$ with the last key in $\boldsymbol{k}$. If they match, the function ver proceeds to verify the previous signature in the sequence, otherwise it tries to verify the signature with the previous key.

Example 1 (2-of-3 multi-signature). Let $\boldsymbol{k} = k_a k_b k_c$, and let $\boldsymbol{\sigma} = \sigma_p \sigma_q$ be such that $\mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathsf{ver}_{k_b}(\sigma_q, \mathsf{T}, 1) = \mathit{true}$, and false otherwise. We have that:

$$\begin{aligned}
\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) &= \mathsf{ver}^{3,2}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } |\boldsymbol{k}| = 3 \text{ and } |\boldsymbol{\sigma}| = 2)\\
&= \mathsf{ver}^{2,2}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_c}(\sigma_q, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}^{1,1}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_b}(\sigma_q, \mathsf{T}, 1) = \mathit{true})\\
&= \mathsf{ver}^{0,0}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathit{true})\\
&= \mathit{true} && (\text{as } m = 0)
\end{aligned}$$

Note that, if we let $\boldsymbol{\sigma'} = \sigma_q \sigma_p$, the resulting evaluation will be:

$$\begin{aligned}
\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) &= \mathsf{ver}^{3,2}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } |\boldsymbol{k}| = 3 \text{ and } |\boldsymbol{\sigma'}| = 2)\\
&= \mathsf{ver}^{2,2}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_c}(\sigma_p, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}^{1,2}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_b}(\sigma_p, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}^{0,1}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathit{true})\\
&= \mathit{false} && (\text{as } m \neq 0 \text{ and } n = 0)
\end{aligned}$$

□

3.4 Semantics of scripts

Definition 7 gives the semantics of script expressions. This semantics will be used in Section 3.5 to define when a transaction can redeem another one. We use an environment $\rho : \mathsf{Var} \rightharpoonup \mathbb{Z}$ which associates a denotation to each variable occurring in it. Further, we use a transaction $\mathsf{T} \in \mathsf{Tx}$ and an index $i \in \mathbb{N}$ to indicate the witness redeeming the script, both used to evaluate the timelock expressions. We use the denotation $\bot$ to represent "failure" of the evaluation. This is the case e.g. of timelock expressions, when the temporal constraint is not satisfied. All the semantic operators used in Definition 7 are *strict*, i.e. they evaluate to $\bot$ if some of their operands is $\bot$.

Definition 7 (Expression evaluation). *Let $\rho : \mathsf{Var} \rightharpoonup \mathbb{Z}$, let $\mathsf{T} \in \mathsf{Tx}$ and $i \in \mathbb{N}$. We define the function $[\![\cdot]\!]_{\mathsf{T},i,\rho} : \mathsf{Exp} \to \mathsf{Den}$ in Figure 3, where we use the following operators on denotations:*

$$\mathit{if}\ \nu_0\ \mathit{then}\ \nu_1\ \mathit{else}\ \nu_2 \equiv \begin{cases} \nu_1 & \mathit{if}\ \nu_0 = \mathit{true} \\ \nu_2 & \mathit{if}\ \nu_0 = \mathit{false} \\ \bot & \mathit{otherwise} \end{cases} \qquad \mathit{size}(\nu) \equiv \begin{cases} \bot & \mathit{if}\ \nu \notin \mathbb{Z} \\ 0 & \mathit{if}\ \nu = 0 \\ \left\lceil \frac{\log_2 |\nu|}{7} \right\rceil & \mathit{otherwise} \end{cases}$$

$$\nu_0 \circ_\bot \nu_1 \equiv \mathit{if}\ \nu_0, \nu_1 \in \mathbb{Z}\ \mathit{then}\ \nu_0 \circ \nu_1\ \mathit{else}\ \bot \qquad (\circ \in \{+, -, =, <\})$$

Definition 8 (Script verification). *We say that the input i of T* verifies *$\lambda \boldsymbol{x}.e$ (in symbols: $\mathsf{T}, i \models \lambda \boldsymbol{x}.e$) when $\boldsymbol{x} = x_1 \dots x_n$, $\mathsf{T}.\mathsf{wit}(i) = k_1 \dots k_n$, and:*

$$[\![e]\!]_{\mathsf{T},i,\{x_j \mapsto k_j \mid j \in 1 \dots n\}} = \mathit{true}$$

$$
\begin{aligned}
[\![x]\!]_{\mathsf{T},i,\rho} &= \rho(x) \\
[\![k]\!]_{\mathsf{T},i,\rho} &= k \\
[\![\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{e})]\!]_{\mathsf{T},i,\rho} &= \mathsf{ver}_{\boldsymbol{k}}([\![\boldsymbol{e}]\!]_{\mathsf{T},i,\rho}, \mathsf{T}, i) \\
[\![\mathsf{H}(e)]\!]_{\mathsf{T},i,\rho} &= H([\![e]\!]_{\mathsf{T},i,\rho}) \qquad (H \text{ is a public hash function}) \\
[\![\mathsf{absAfter}\ t : e]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ \mathsf{T}.\mathsf{absLock} \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot \\
[\![\mathsf{relAfter}\ t : e]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ \mathsf{T}.\mathsf{relLock}(i) \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot \\
[\![e \circ e']\!]_{\mathsf{T},i,\rho} &= [\![e]\!]_{\mathsf{T},i,\rho} \circ_{\bot} [\![e']\!]_{\mathsf{T},i,\rho} \qquad (\circ \in \{+, -, =, <\}) \\
[\![|e|]\!]_{\mathsf{T},i,\rho} &= \mathit{size}([\![e]\!]_{\mathsf{T},i,\rho}) \\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ [\![e_0]\!]_{\mathsf{T},i,\rho}\ \mathit{then}\ [\![e_1]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ [\![e_2]\!]_{\mathsf{T},i,\rho}
\end{aligned}
$$

Fig. 3: Semantics of script expressions.

Example 2. Let H be a hash function, let $s, h \in \mathbb{Z}$ be such that $h = H(s)$, and let T be such that $\mathsf{T}.\mathsf{wit}(1) = (\sigma, s)$, with $\sigma = \mathsf{sig}_k^{aa}(\mathsf{T})$. We prove that:

$$\mathsf{T}, 1 \models \lambda(\varsigma, x).\big(\mathsf{versig}_k(\varsigma)\ \mathsf{and}\ \mathsf{H}(x) = h\big)$$

To do this, let $\rho = \{\varsigma \mapsto \sigma, x \mapsto s\}$. We have that:

$$
\begin{aligned}
&[\![\mathsf{versig}_k(\varsigma)\ \mathsf{and}\ \mathsf{H}(x) = h]\!]_{\mathsf{T},1,\rho} \;=\; [\![\mathsf{versig}_k(\varsigma)]\!]_{\mathsf{T},1,\rho}\ \mathit{and}\ [\![\mathsf{H}(x) = h]\!]_{\mathsf{T},1,\rho} \\
&= \mathsf{ver}_k([\![\varsigma]\!]_{\mathsf{T},1,\rho}, \mathsf{T}, 1)\ \mathit{and}\ ([\![\mathsf{H}(x)]\!]_{\mathsf{T},1,\rho} =_{\bot} [\![h]\!]_{\mathsf{T},1,\rho}) \\
&= \mathsf{ver}_k(\rho(\varsigma), \mathsf{T}, 1)\ \mathit{and}\ (H([\![x]\!]_{\mathsf{T},1,\rho}) =_{\bot} h) \;=\; \mathsf{ver}_k(\sigma, \mathsf{T}, 1)\ \mathit{and}\ (H(\rho(x)) =_{\bot} h) \\
&= \mathit{true}
\end{aligned}
$$

□

3.5 Semantics of transactions

Definition 9 describes when the j-th input of a transaction T' (put on the blockchain at time t') can redeem v Satoshis from the i-th output of the transaction T (put on the blockchain at time t). We denote this by $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Definition 9 (Output redeeming). *We write* $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$ *iff all the following conditions hold:*

(a) $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{T}\{\mathsf{wit} \mapsto \bot\}, i)$
(b) $\mathsf{T}', j \models \mathit{script}(\mathsf{T}.\mathsf{out}(i))$
(c) $v = \mathit{val}(\mathsf{T}.\mathsf{out}(i))$
(d) $t' \geq \mathsf{T}'.\mathsf{absLock}$
(e) $t' - t \geq \mathsf{T}'.\mathsf{relLock}(j) \quad$ *if* $j \in \mathrm{dom}\, \mathsf{T}'.\mathsf{relLock}$

We write $(\mathsf{T}, i, t) \not\rightsquigarrow (\mathsf{T}', j, t')$ *when for no* v *it holds that* $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Item (a) links the j-th input of T' to the i-th output of T. Note that, since we are modelling SegWit, the witness in the transaction $\mathsf{T}'.\mathsf{in}(j)$ is left unspecified:

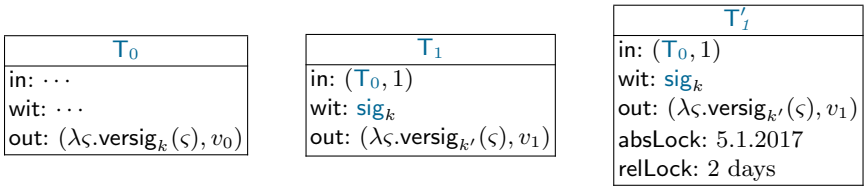

Fig. 4: Three transactions. For notational conciseness, when displaying transactions we omit the substitution $\{\mathsf{wit} \mapsto \bot\}$ for the transaction within the in field (e.g., we just write T_0 within $\mathsf{T}_1.\mathsf{in}$). Also, we use dates in time constraints.

this is why we set to $\bot$ also the witness of T. Item (b) requires that the j-th witness of T' verifies the i-th output script of T. Item (c) just defines v as the value in the i-th output of T. Items (d) and (e) check the absolute and relative timelocks, respectively. The first constraint states that T' cannot appear on the blockchain before $\mathsf{T}'.\mathsf{absLock}$; the second one states that T' cannot appear until at least $\mathsf{T}'.\mathsf{relLock}(j)$ time units have elapsed since T was put on the blockchain.

Example 3. With the transactions in Figure 4, we have $(\mathsf{T}_0, 1, t_0) \overset{v_0}{\rightsquigarrow} (\mathsf{T}_1, 1, t_1)$. Indeed, for item (a) we have that $\mathsf{T}_1.\mathsf{in}(1) = (\mathsf{T}_0\{\mathsf{wit} \mapsto \bot\}, 1)$; for item (b), $\mathsf{T}_1, 1 \models \lambda\varsigma.\mathsf{versig}_k(\varsigma)$; for item (c), $v_0 = \mathit{val}(\mathsf{T}_0.\mathsf{out}(1))$. The other two items trivially hold, as there are no time constraints. We also have $(\mathsf{T}_0, 1, 2.1.2017) \overset{v_0}{\rightsquigarrow} (\mathsf{T}'_1, 1, 6.1.2017)$. To show that, we have to check also items (d) and (e). For item (d), we have that $6.1.2017 \geq \mathsf{T}'_1.\mathsf{absLock} = 5.1.2017$. For item (e), we have that $6.1.2017 - 2.1.2017 \geq \mathsf{T}'_1.\mathsf{relLock}(1) = 2$ days. □

3.6 Blockchain and consistency

In Definition 10 we model blockchains as sequences of *timed transactions* (T, t), where t represents the time when the transaction T has been added. Note that our definition is very permissive: for instance, it allows a blockchain to contain transactions which do not redeem any transactions, or double-spent transactions. We will rule out such *inconsistent* blockchains later on in Definition 13.

Definition 10 (Blockchain). *A blockchain* $\mathbf{B}$ *is a sequence* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$, *where* T_1 *is the only transaction with* $\mathsf{in} = \bot$, *and* $t_i \leq t_j$ *for all* $1 \leq i \leq j \leq n$. *We denote with* $\mathit{trans}_{\mathbf{B}}$ *the set of transactions occurring in* $\mathbf{B}$, *and with* $\mathit{time}_{\mathbf{B}}(\mathsf{T}_i)$ *the time* t_i *of transaction* T_i *in* $\mathbf{B}$. *Given a transaction* T, *we define* $\mathit{match}_{\mathbf{B}}(\mathsf{T})$ *as the set of transactions* T_i *such that* $\mathsf{T}\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}_i\{\mathsf{wit} \mapsto \bot\}$.

Definition 11 (Unspent output). *Let* $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *be a blockchain. We say that the output* j *of transaction* T_i *is* unspent *in* $\mathbf{B}$ *whenever:*

$$\forall i' \leq n, j' \in \mathbb{N} \;\; : \;\; (\mathsf{T}_i, j, t_i) \not\rightsquigarrow (\mathsf{T}_{i'}, j', t_{i'})$$

Given a blockchain $\mathbf{B}$, *we define:*

T_1	
in:	$\bot$
wit:	$\bot$
out:	$1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_1}(\varsigma), 3)$
	$2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_2}(\varsigma), 5)$
	$3 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_3}(\varsigma), 7)$

T_2	
in:	$1 \mapsto (\mathsf{T}_1, 2), 2 \mapsto (\mathsf{T}_1, 3)$
wit:	$1 \mapsto sig_{k_2}, 2 \mapsto sig_{k_3}$
out:	$(\lambda\varsigma.\mathsf{versig}_{k_2}(\varsigma), 10)$

T_3	
in:	$(\mathsf{T}_1, 2)$
wit:	sig_{k_2}
out:	$(\lambda\varsigma.\mathsf{versig}_{k_2}(\varsigma), 5)$

Fig. 5: Three transactions for Examples 4 to 6.

- *$UTXO_{\mathbf{B}}$, the* Unspent Transaction Output *of* $\mathbf{B}$*, as the set of pairs* (T_i, j) *such that output* j *of* T_i *is unspent in* $\mathbf{B}$*.*
- *$val(\mathbf{B})$, the* value *of* $\mathbf{B}$*, as the sum of the values of all outputs in its UTXO.*

Example 4. Consider the transactions in Figure 5, and let $\mathbf{B} = (\mathsf{T}_1, 0)(\mathsf{T}_2, t_2)$. We have that $(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2)$ and $(\mathsf{T}_1, 3, 0) \overset{7}{\rightsquigarrow} (\mathsf{T}_2, 2, t_2)$, while the other outputs are unspent. Hence, the UTXO of $\mathbf{B}$ is $\{(\mathsf{T}_1, 1), (\mathsf{T}_2, 1)\}$. □

The following definition establishes when (T, t) is a *consistent update* of $\mathbf{B}$.

Definition 12 (Consistent update). *We write* $\mathbf{B} \rhd (\mathsf{T}, t)$ *iff either* $\mathbf{B} = []$*,* T *is initial and* $t = 0$*, or, given, for all* $i \in \mathrm{dom}\,(\mathsf{T}.\mathsf{in})$*:*

$$
\begin{array}{ll}
\{\mathsf{T}'_i\} = match_{\mathbf{B}}(fst(\mathsf{T}.\mathsf{in}(i))) & \textit{(redeemed transaction)} \\
o_i = snd(\mathsf{T}.\mathsf{in}(i)) & \textit{(redeemed output index)} \\
t'_i = time_{\mathbf{B}}(\mathsf{T}'_i) & \textit{(time when } \mathsf{T}'_i \textit{ was added to } \mathbf{B}\textit{)} \\
v_i = val(\mathsf{T}'_i.\mathsf{out}(o_i)) & \textit{(value of the redeemed output)}
\end{array}
$$

the following conditions hold:

(1) $\forall i \in \mathrm{dom}\,\mathsf{T}.\mathsf{in} : (\mathsf{T}'_i, o_i) \in UTXO_{\mathbf{B}}$

(2) $\forall i \in \mathrm{dom}\,\mathsf{T}.\mathsf{in} : (\mathsf{T}'_i, o_i, t'_i) \overset{v_i}{\rightsquigarrow} (\mathsf{T}, i, t)$

(3) $\sum \{v_i \mid i \in \mathrm{dom}\,\mathsf{T}.\mathsf{in}\} \;\geq\; \sum \{val(\mathsf{T}.\mathsf{out}(j)) \mid j \in \mathrm{dom}\,\mathsf{T}.\mathsf{out}\}$

(4) $\mathbf{B} = \mathbf{B}'(\mathsf{T}', t') \implies t \geq t'$

Firstly, for each $\mathsf{T}.\mathsf{in}(i)$ we obtain the singleton $\{\mathsf{T}'_i\}$ from the blockchain, using $match_{\mathbf{B}}$, such that $fst(\mathsf{T}.\mathsf{in}(i))\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}'_i\{\mathsf{wit} \mapsto \bot\}$. The update is inconsistent if $match_{\mathbf{B}}(fst(\mathsf{T}.\mathsf{in}(i)))$ is not a singleton for some i. Condition (1) requires that the redeemed outputs are currently unspent in $\mathbf{B}$. Condition (2) asks that each input of T redeems an output of a transaction in $\mathbf{B}$. Condition (3) requires that the sum of the values of the outputs of T is not greater than the total value it redeems. Finally, (4) requires that the time of T is greater than or equal to the time of the last transaction in $\mathbf{B}$.

Example 5. Consider again the transactions in Figure 5, and let $\mathbf{B} = (\mathsf{T}_1, 0)$. We prove that $\mathbf{B} \rhd (\mathsf{T}_2, t_2)$. Let $o_1 = 2$, $o_2 = 3$, $t'_1 = t'_2 = 0$, $v_1 = 5$, $v_2 = 7$. We now prove that the conditions of Definition 12 are satisfied. For condition (1), note that both $(\mathsf{T}_1, 2)$ and $(\mathsf{T}_1, 3)$ are unspent, according to Definition 11. For condition (2), note that:

$$(\mathsf{T}_1, 2, 0) \overset{v_1}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2) \qquad (\mathsf{T}_1, 3, 0) \overset{v_2}{\rightsquigarrow} (\mathsf{T}_2, 2, t_2)$$

hold, according to Definition 9. Finally, for condition (3), we have that:

$$\sum \{v_i \mid i \in \{1, 2\}\} = 5 + 7 \;\geq\; \sum \{val(\mathsf{T}_2.\mathsf{out}(j)) \mid j \in \operatorname{dom} \mathsf{T}_2.\mathsf{out}\} = 10$$

Therefore, (T_2, t_2) is a consistent update of $\mathbf{B}$. □

Example 6 (Double spending). Consider again the transactions in Figure 5, and let $\mathbf{B} = (\mathsf{T}_1, 0)(\mathsf{T}_2, t_2)$. We prove that (T_3, t_3) is *not* a consistent update of $\mathbf{B}$. Although condition (2) of Definition 12 holds:

$$(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_3, 1, t_3)$$

we have that condition (1) is *not* satisfied. In fact, according to Definition 11, $(\mathsf{T}_1, 2)$ is already spent in $\mathbf{B}$ because

$$(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2)$$

holds and both T_1 and T_2 are in $\mathbf{B}$. Since T_3 is trying to spend an output already spent, this transaction should not be appended to $\mathbf{B}$. □

We now define when a blockchain is consistent. Intuitively, consistency holds when the blockchain has been constructed, started from the empty one, by appending consistent updates, only. The actual definition is given by induction.

Definition 13 (Consistency). *We say that a blockchain* $\mathbf{B}$ *is consistent if either* $\mathbf{B} = []$*, or* $\mathbf{B} = \mathbf{B}'(\mathsf{T}, t)$ *with* $\mathbf{B}'$ *consistent and* $\mathbf{B}' \rhd (\mathsf{T}, t)$*.*

Note that the empty blockchain is consistent; the blockchain with a single transaction (T_1, t_1) is consistent iff T_1 is initial and $t_1 = 0$. The transaction T_1 models the first transaction in the *genesis block* (as discussed in Section 6, we are abstracting away the *coinbase* transactions, which forge new bitcoins).

We now establish some basic properties of consistent blockchains. Lemma 1 states that, in a consistent blockchain, the inputs of a transaction point backwards to the output of some transaction in the blockchain.

Lemma 1. *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

$$\forall i \in 2 \ldots n : \forall (\mathsf{T}, h) \in \operatorname{ran}(\mathsf{T}_i.\mathsf{in}) : \exists j < i : \mathsf{T}_j\{\mathsf{wit} \mapsto \bot\} = \mathsf{T} \wedge h \in \operatorname{dom}(\mathsf{T}_j.\mathsf{out})$$

The following theorem establishes that a transaction output cannot be redeemed twice in a consistent blockchain.

Theorem 1 (No double spending). *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

$$\forall i \neq j \in 1 \ldots n \;\; : \;\; \mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \;\cap\; \mathrm{ran}\,(\mathsf{T}_j.\mathsf{in}) \;=\; \emptyset$$

The following lemma states that there can be at most a single match of an arbitrary transaction within a consistent blockchain. This implies that the in field of an arbitrary transaction points at most to one transaction output within the blockchain.

Lemma 2. *If* $\mathbf{B}$ *is consistent, then for all transactions* T*,* $match_{\mathbf{B}}(\mathsf{T})$ *contains at most one element.*

Lemma 3 ensures that all the transactions on a consistent blockchain are pairwise distinct, even when neglecting their witnesses.

Lemma 3. *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

$$\forall i \neq j \in 1 \ldots n \;\; : \;\; \mathsf{T}_i\{\mathsf{wit} \mapsto \bot\} \neq \mathsf{T}_j\{\mathsf{wit} \mapsto \bot\}$$

The following theorem states that the overall value of a blockchain decreases as the blockchain grows. This is because our model does not keep track of the *coinbase transactions*, which in Bitcoin allow miners to collect transaction fees (the difference between inputs and outputs of a transaction), and block rewards.

Theorem 2 (Non-increasing value). *Let* $\mathbf{B}$ *be a consistent blockchain, and let* $\mathbf{B}'$ *be a non-empty prefix of* $\mathbf{B}$*. Then,* $val(\mathbf{B}') \geq val(\mathbf{B})$*.*

Note that the scripting language and its semantics are immaterial in all the statements above. Actually, proving these results never involves checking condition (b) of Definition 9. Of course, the choice of the scripting language affects the expressiveness of the smart contracts built upon Bitcoin.

4 Example: static chains of transactions

We now formally specify in our model a simple smart contract[7], which illustrates the impact of SegWit on the expressiveness of Bitcoin contracts.

A participant A wants to send an indirect payment of 1฿ to C, routing it through B. To authorize the payment, B wants to keep a fee of 0.1฿. However, A is afraid that B will keep all the money for himself, so she exploits the following contract. She creates a *chain* of transactions, as shown in Figure 6. The transaction T_{AB} transfers 1.1฿ from A to B (but it is not signed by A, yet), while T_{BC} transfers 1฿ from B to C. We assume that $(\mathsf{T}_{\mathsf{A}}, 1)$ is a transaction output redeemable by A through her key k_{A}, and that k_{B} is the key of B.

The protocol of A is the following: A starts by asking B for his signature on T_{BC}, ensuring that C will be paid. After receiving and verifying the signature,

[7] https://www.bitcoinhk.org/media/presentations/2016-03-16/2016-03-16-Segregated_Witness.pdf

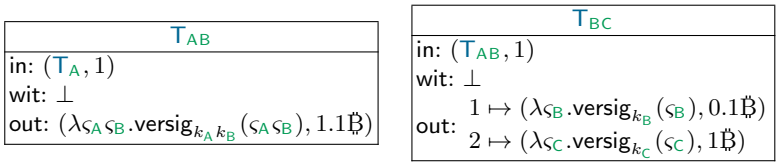

Fig. 6: Transactions of the chain contract.

A puts T_{AB} on the blockchain, adding her signature on the wit field. Then, she also appends T_{BC}, replacing the wit field with her signature and B's one. Since A takes care of publishing the transactions, the behaviour of B consists just in sending his signature on T_{BC}.

Remarkably, this contract relies on the SegWit feature: indeed, without SegWit it no longer works. We can disable SegWit by changing our model as follows:

- in Definition 2, we no longer require that $\forall i \in \mathrm{dom}\,\mathsf{in} : \mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$
- in Definition 9, we replace item (a) with the condition: $\mathsf{T}'.\mathsf{in}(j) \;=\; (\mathsf{T}, i)$
- in Definition 10, we let $\mathit{match}_{\mathbf{B}}(\mathsf{T}) = \{\mathsf{T}\}$ if T occurs in **B**, empty otherwise.

To see why disabling SegWit breaks the contract, assume that the transaction $\mathsf{T} = \mathsf{T}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{AB}})\}$ is unspent on the blockchain, when participant A attempts to append also $\mathsf{T}' = \mathsf{T}_{\mathsf{BC}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{BC}})\,\mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}_{\mathsf{BC}})\}$. To be a consistent update, by item (2) of Definition 12 we must have (for some $t_1 \le t_2$):

$$(\mathsf{T}, 1, t_1) \overset{1\text{Ƀ}}{\rightsquigarrow} (\mathsf{T}', 1, t_2) \tag{1}$$

For this, all the conditions in Definition 9 must hold. However, since we have disabled SegWit, for item (a) we no longer check that:

$$\mathsf{T}'.\mathsf{in}(1) \;=\; (\mathsf{T}\{\mathsf{wit} \mapsto \bot\}, 1)$$

but instead we need to check the condition:

$$\tilde{\mathsf{T}}'.\mathsf{in}(1) \;=\; (\tilde{\mathsf{T}}, 1) \tag{2}$$

where the transactions $\tilde{\mathsf{T}}, \tilde{\mathsf{T}}'$ correspond to the non-SegWit versions of T, T', i.e. their in fields point to their actual parents, according to the new Definition 2.

Hence, condition (2) checks the equality between $\tilde{\mathsf{T}}_{\mathsf{AB}}$ (the transaction in the input of $\tilde{\mathsf{T}}'$) and $\tilde{\mathsf{T}}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\tilde{\mathsf{T}}_{\mathsf{AB}})\}$ (the transaction $\tilde{\mathsf{T}}$). Note that all the fields of the second transaction — but the wit field — are equal to those of the first transaction. Instead, the witness of $\tilde{\mathsf{T}}_{\mathsf{AB}}$ is $\bot$, while the one of $\tilde{\mathsf{T}}$ contains the signature of A. This difference in the wit field is ignored with the SegWit semantics, while it is discriminating for the older version of Bitcoin.

A naïve attempt to amend the contract would be to set the input field of $\tilde{\mathsf{T}}'$ to $\tilde{\mathsf{T}}$. However, this would invalidate the signature of A on $\tilde{\mathsf{T}}'$.

5 Compiling to standard Bitcoin transactions

We now sketch how to compile the transactions of our abstract model into concrete Bitcoin transactions. In particular, we aim at producing *standard* Bitcoin transactions, which respect further constraints on their fields[8]. This is crucial, because non-standard transactions are mostly discarded by the Bitcoin network.

Our compiler produces output scripts of the following kinds, which are all allowed in standard transactions:

Pay to Public Key Hash (P2PKH) takes as parameters a public key and a signature, and checks that (i) the hash of the public key matches the hash hardcoded in the script; (ii) the signature is verified against the public key.

Pay to Script Hash (P2SH) contains only a hash (say, h). The actual script $\lambda \boldsymbol{x}.e$ — which is *not* required to be standard — is contained instead in the wit field of the redeeming transaction, alongside with the actual parameters $\boldsymbol{k}$. The evaluation succeeds if $H(\lambda \boldsymbol{x}.e) = h$ and $(\lambda \boldsymbol{x}.e)\boldsymbol{k}$ evaluates to *true*. The only constraint imposed by P2SH is on the size of the script, which is limited to the size of a stack element (520 bytes).

OP_RETURN allows to put up to 80 bytes of data in an output script, making the output unredeemable.

We compile the scripts of the form $\lambda \varsigma.\mathsf{versig}_k(\varsigma)$ to P2PKH, and those of the form $\lambda.k$ to OP_RETURN. All other scripts are compiled to P2SH when they comply with the size constraint, otherwise compilation fails. In this way, our compiler always produces standard transactions.

Our compiler exploits the *alternative stack* as temporary storage of the variable values. In this way we cope with the stack-based nature of the Bitcoin scripting language. For instance, for the script $\lambda x.\mathsf{H}(x) = \mathsf{H}(x+1)$, the variable x is pushed on the alternative stack beforehand, then duplicated and copied in the main stack before each operation involving x.

6 Conclusions

We have proposed a formal model for Bitcoin transactions. Our model abstractly describes their essential aspects, at the same time enabling formal reasoning, and providing a formal specification to some of Bitcoin's less documented features.

An alternative model of transactions in blockchain systems has been proposed in [7]. Roughly, blockchains are represented as directed acyclic graphs, where edges denote transfers of assets. This model is quite abstract, so that it can be instantiated to different blockchains (e.g., Bitcoin, Ethereum, and Hyperledger Fabric). Differently from ours, the model in [7] does not capture some peculiar features of Bitcoin, like e.g. transaction signatures and signature modifiers, output scripts, multi-signature verification, and Segregated Witnesses.

[8] https://bitcoin.org/en/developer-guide#standard-transactions

Our work provides the theoretical foundations to model Bitcoin smart contracts, reducing the gap between cryptography and programming languages communities. A formal description of smart contracts would enable their automated verification and analysis, which are of crucial importance in a context where bugs in design or implementation may result in loss of money.

Differences between our model and Bitcoin There are some differences between our model and the actual Bitcoin, which we outline below.

In Definition 2, we stipulate that the in field of a transaction points to another transaction. Instead, in Bitcoin the in field contains the identifier of the input transaction. More specifically, this identifier is defined as $H(\mu(\mathsf{T}))$, where:

- $\mu = \{\mathsf{wit} \mapsto \bot\}$ since the activation of the SegWit feature;
- $\mu = \bot$, beforehand.

Consequently, the condition $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$ item (a) of Definition 9 would be translated in Bitcoin as: $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{H}(\mu(\mathsf{T}'')), i)$, where $\mathsf{H}(\mu(\mathsf{T}'')) = \mathsf{H}(\mu(\mathsf{T}))$. Intuitively, the in field specifies the transaction (and the output index) to redeem. Since the activation of SegWit, the computation of the transaction identifier does not take in account the wit field.

The scripting language in Definition 1 is a bit more expressive than Bitcoin's. For instance, the script $\lambda x.\mathsf{H}(x) < k$ is admissible in our model, while it is not in Bitcoin. Indeed, the Bitcoin scripting language only admits the comparison (via the `OP_LESSTHANOREQUAL` opcode) on 32-bit integers, while two arbitrary values can only be tested for equality (via the `OP_EQUAL` opcode). Similar restrictions apply to arithmetic operations. It is straightforward to adapt our model to apply the same restrictions on Bitcoin scripts. Indeed, our compiler already implements a simple type system which rules away scripts not admissible in Bitcoin.

Definition 10 represents blockchains as sequences of transactions. Instead, in Bitcoin they are sequences of *blocks* of transactions. In this way, we are abstracting both from the cryptographic puzzle that miners have to solve to append a new block to the blockchain, and from the *coinbase transactions*, which (like our initial transaction) do not redeem other transactions, and mint new bitcoins (the block rewards). Coinbase transactions are also used in Bitcoin to collect transaction fees, which are just discarded in our model. Note that extending our model with coinbase transactions would falsify Theorem 2, since the overall value in the blockchain would no longer be decreasing.

In Definitions 2 and 9, the $\mathsf{absLock}$ and $\mathsf{relLock}$ fields specify the time when a transaction can be appended to the blockchain. In Bitcoin transactions, besides the time we can also use the *block height*, i.e. the distance between any given block and the genesis block. Setting the block height to h implies that the transaction can be mined from the block h onward.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia project P.I.A. 2013 "NOMAD". Stefano Lande gratefully acknowledges Sardinia Regional Government for the financial support of his PhD scholarship (P.O.R.

Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2014-2020).

References

1. 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)
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE Symposium on Security and Privacy. pp. 443–458 (2014)
3. 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)
4. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017)
5. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014)
6. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In: IEEE S & P. pp. 104–121 (2015)
7. Cachin, C., Caro, A.D., Moreno-Sanchez, P., Tackmann, B., Vukolić, M.: The transaction graph for modeling blockchain semantics. Cryptology ePrint Archive, Report 2017/1070 (2017), https://eprint.iacr.org/2017/1070
8. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014)
9. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015)
10. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops. pp. 4–13 (2017)
11. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography Workshops. pp. 126–141. Springer (2016)
12. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
13. O'Connor, R., Piekarska, M.: Enhancing Bitcoin transactions with covenants. In: Financial Cryptography Workshops (2017)
14. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997), http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548

A Proofs

Proof of Lemma 1

By Definition 13, (T_i, t_i) is a consistent update of $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_{i-1}, t_{i-1})$. The thesis follows from condition (2) of Definition 12. □

Proof of Theorem 1

Let $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ be consistent. By contradiction, assume that there exist $i < j$ and i', j' such that $\mathsf{T}_i.\mathsf{in}(i') = \mathsf{T}_j.\mathsf{in}(j')$. By consistency, there exist h, h' such that $(\mathsf{T}_h\{\mathsf{wit} \mapsto \bot\}, h') = \mathsf{T}_i.\mathsf{in}(i')$. Since $\mathbf{B}_{1..i-1} \rhd (\mathsf{T}_i, t_i)$, then by item (2) of Definition 12 it must be $(\mathsf{T}_h, h', t_h) \rightsquigarrow (\mathsf{T}_i, i', t_i)$. Hence, by Definition 11 it follows that (T_h, h') is already *spent* in $\mathbf{B}$. Since $\mathbf{B}_{1..j-1} \rhd (\mathsf{T}_j, t_j)$, by item (1) of Definition 12, (T_h, h') must be *unspent* — contradiction. □

Proof of Lemma 2

Let $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ be consistent. By contradiction, assume that $\mathsf{T}_i, \mathsf{T}_j \in \mathit{match}_{\mathbf{B}}(\mathsf{T})$, with $\mathsf{T}_i \neq \mathsf{T}_j$ (and so, $i \neq j$). By Definition 10 it must be $\mathsf{T}_i\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}_j\{\mathsf{wit} \mapsto \bot\}$, hence in particular $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in}$. There are two cases. If $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in} = \bot$, then by Definition 10 $\mathbf{B}$ is not a blockchain, since $i \neq j$. Hence, $\mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \cap \mathrm{ran}\,(\mathsf{T}_j.\mathsf{in}) = \mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \neq \emptyset$. By Theorem 1, this cannot happen because $\mathbf{B}$ is consistent — contradiction. □

Proof of Lemma 3

Straightforward from Lemma 2, taking $\mathsf{T} = \mathsf{T}_j$. □

Proof of Theorem 2

Let $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$. By contradiction, there exists some $i < n$ such that, given $\mathbf{B}_i = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_i, t_i)$:

$$\mathit{val}(\mathbf{B}_i) \;<\; \mathit{val}(\mathbf{B}_i(\mathsf{T}_{i+1}, t_{i+1}))$$

Let U_i and U_{i+1} be the UTXOs of $\mathbf{B}_i$ and of $\mathbf{B}_i(\mathsf{T}_{i+1}, t_{i+1})$, respectively, and let $U = U_i \cap U_{i+1}$. Since $\mathit{val}(U_i) < \mathit{val}(U_{i+1})$, then it must be $\mathit{val}(U_i \setminus U) < \mathit{val}(U_{i+1} \setminus U)$. The set $U_i \setminus U$ contains the outputs redeemed by T_{i+1}, while the set $U_{i+1} \setminus U$ contains exactly the outputs in T_{i+1}. Since $\mathbf{B}$ is consistent, then $\mathbf{B}_i \rhd (\mathsf{T}_{i+1}, t_{i+1})$. Then, by Definition 12, for each $k \in \mathrm{dom}\,\mathsf{T}_{i+1}.\mathsf{in}$, there exists a unique $j \leq i$ such that, given $o_k = \mathit{snd}(\mathsf{T}_{i+1}.\mathsf{in}(k))$ and $v_k = \mathit{val}(\mathsf{T}_j.\mathsf{out}(o_k))$:

$$(\mathsf{T}_j, o_k, t_j) \overset{v_k}{\rightsquigarrow} (\mathsf{T}_{i+1}, k, t_{i+1})$$

Then, by item (3) of Definition 12:

$$\begin{aligned} \mathit{val}(U_i \setminus U) &= \textstyle\sum \{v_k \mid k \in \mathrm{dom}\,\mathsf{T}_{i+1}.\mathsf{in}\} \\ &\geq \textstyle\sum \{\mathit{val}(\mathsf{T}_{i+1}.\mathsf{out}(h)) \mid h \in \mathrm{dom}\,\mathsf{T}_{i+1}.\mathsf{out}\} = \mathit{val}(U_{i+1} \setminus U) \end{aligned}$$

while we assumed $\mathit{val}(U_i \setminus U) < \mathit{val}(U_{i+1} \setminus U)$ — contradiction. □