SoK: unraveling Bitcoin smart contracts

Nicola Atzei[1], Massimo Bartoletti[1], Tiziana Cimoli[1], Stefano Lande[1], Roberto Zunino[2]

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

Abstract. Albeit the primary usage of Bitcoin is to exchange currency, its blockchain and consensus mechanism can also be exploited to securely execute some forms of *smart contracts*. These are agreements among mutually distrusting parties, which can be automatically enforced without resorting to a trusted intermediary. Over the last few years a variety of smart contracts for Bitcoin have been proposed, both by the academic community and by that of developers. However, the heterogeneity in their treatment, the informal (often incomplete or imprecise) descriptions, and the use of poorly documented Bitcoin features, pose obstacles to the research. In this paper we present a comprehensive survey of smart contracts on Bitcoin, in a uniform framework. Our treatment is based on a new formal specification language for smart contracts, which also helps us to highlight some subtleties in existing informal descriptions, making a step towards automatic verification. We discuss some obstacles to the diffusion of smart contracts on Bitcoin, and we identify the most promising open research challenges.

1 Introduction

The term "smart contract" was conceived in [43] to describe agreements between two or more parties, that can be automatically enforced without a trusted intermediary. Fallen into oblivion for several years, the idea of smart contract has been resurrected with the recent surge of distributed ledger technologies, led by Ethereum and Hyperledger. In such incarnations, smart contracts are rendered as computer programs. Users can request the execution of contracts by sending suitable *transactions* to the nodes of a peer-to-peer network. These nodes collectively maintain the history of all transactions in a public, append-only data structure, called *blockchain*. The sequence of transactions on the blockchain determines the state of each contract, and, accordingly, the assets of each user.

A crucial feature of smart contracts is that their correct execution does *not* rely on a trusted authority: rather, the nodes which process transactions are assumed to be mutually untrusted. Potential conflicts in the execution of contracts are resolved through a *consensus* protocol, whose nature depends on the specific platform (e.g., it is based on "proof-of-work" in Ethereum). Ideally, contracts execute correctly whenever the adversary does not control the majority of some resource (e.g., computational power for "proof-of-work" consensus).

The absence of a trusted intermediary, combined with the possibility of transferring money given by blockchain-based cryptocurrencies, creates a fertile ground for the development of smart contracts. For instance, a smart contract may promise to pay a reward to anyone who provides some value that satisfies a given public predicate. This generalises cryptographic puzzles, like breaking a cipher, inverting a hash function, etc.

Since smart contracts handle the ownership of valuable assets, attackers may be tempted to exploit vulnerabilities in their implementation to steal or tamper with these assets. Although analysis tools [17,30,34] may improve the security of contracts, so far they have not been able to completely prevent attacks. For instance, a series of vulnerabilities in Ethereum contracts [10] have been exploited, causing money losses in the order of hundreds of millions of dollars [3–5].

Using domain-specific languages (possibly, not Turing-complete) could help to overcome these security issues, by reducing the distance between contract specification and implementation. For instance, despite the discouraging limitations of its scripting language, Bitcoin has been shown to support a variety of smart contracts. Lotteries [6, 14, 16, 36], gambling games [32], contingent payments [13, 24, 35], and other kinds of fair multi-party computations [8, 31] are some examples of the capabilities of Bitcoin as a smart contracts platform.

Unlike Ethereum, where contracts can be expressed as computer programs with a well-defined semantics, Bitcoin contracts are usually realised as cryptographic protocols, where participants send/receive messages, verify signatures, and put/search transactions on the blockchain. The informal (often incomplete or imprecise) narration of these protocols, together with the use of poorly documented features of Bitcoin (e.g., segregated witnesses, scripts, signature modifiers, temporal constraints), and the overall heterogeneity in their treatment, pose serious obstacles to the research on smart contracts in Bitcoin.

Contributions. This paper is, at the best of our knowledge, the first systematic survey of smart contracts on Bitcoin. In order to obtain a uniform and precise treatment, we exploit a new formal model of contracts. Our model is based on a process calculus with primitives to construct Bitcoin transactions, to put them on the blockchain, and to search the blockchain for transactions matching given patterns. Our calculus allows us to give smart contracts a precise operational semantics, which describes the interactions of the (possibly dishonest) participants involved in the execution of a contract.

We exploit our model to systematically formalise a large portion of the contracts proposed so far both by researchers and Bitcoin developers. In many cases, we find that specifying a contract with the intended security properties is significantly more complex than expected after reading the informal descriptions of the contract. Usually, such informal descriptions focus on the case where all participants are honest, neglecting the cases where one needs to compensate for some unexpected behaviour of the dishonest environment.

Overall, our work aims at building a bridge between research communities: from that of cryptography, where smart contracts have been investigated first, to those of programming languages and formal methods, where smart contracts

could be expressed using proper linguistic models, supporting advanced analysis and verification techniques. We outline some promising research perspectives on smart contracts, both in Bitcoin and in other cryptocurrencies, where the synergy between the two communities could have a strong impact in future research.

2 Background on Bitcoin transactions

In this section we give a minimalistic introduction to Bitcoin [21,38], focussing on the crucial notion of transaction. To this purpose, we rely on the model of Bitcoin transactions in [11]. Here, instead of repeating the formal machinery of [11], we introduce the needed concepts through a series of examples. We will however follow the same notation of [11], and point to the formal definitions therein, to allow the reader to make precise the intuitions provided in this paper.

Bitcoin is a decentralised infrastructure to securely transfer currency (the *bitcoins*, ₿) between users. Transfers of bitcoins are represented as *transactions*, and the history of all transactions is stored in a public, append-only, distributed data structure called *blockchain*. Each user can create an arbitrary number of pseudonyms through which sending and receiving bitcoins. The balance of a user is not explicitly stored within the blockchain, but it is determined by the amount of unspent bitcoins directed to the pseudonyms under her control, through one or more transactions. The logic used for linking inputs to outputs is specified by programmable functions, called *scripts*.

Hereafter we will abstract from a few technical details of Bitcoin, e.g. the fact that transactions are grouped into blocks, and that each transaction must pay a fee to the "miner" who appends it to the blockchain. We refer to [11] for a discussion on the differences between the formal model and the actual Bitcoin.

2.1 Transactions

In their simplest form, Bitcoin transactions allow to transfer bitcoins from one participant to another one. The only exception are the so-called *coinbase* transactions, which can generate fresh bitcoins. Following [11], we assume that there exists a single coinbase transaction, the first one in the blockchain. We represent this transaction, say T_0, as follows:

T_0
in: $\bot$
wit: $\bot$
out: $(\lambda x.\, x < 51, 1฿)$

The transaction T_0 has three fields. The fields in and wit are set to $\bot$, meaning that T_0 does not point backwards to any other transaction (since T_0 is the first one on the blockchain). The field out contains a pair. The first element of the pair, $\lambda x.\, x < 51$, is a *script*, that given as input a value x, checks if $x < 51$ (this is just for didactical purposes: we will introduce more useful scripts in a while). The second element of the pair, 1฿, is the amount of currency that can be transferred to other transactions.

Now, assume that participant A wants to *redeem* 1Ƀ from T_0, and transfer that amount under her control. To do this, A has to append to the blockchain a new transaction, e.g.:

T_A
in: T_0
wit: 42
out: $(\lambda x.\mathsf{versig}_{k_\mathsf{A}}(x), 1Ƀ)$

The field in points to the transaction T_0 in the blockchain. To be able to redeem from there 1Ƀ, A must provide a *witness* which makes the script within T_0.out evaluate to true. In this case the witness is 42, hence the redeem succeeds, and T_0 is considered *spent*. The script within T_A.out is the most commonly used one in Bitcoin: it verifies the signature x with A's public key. The message against which the signature is verified is the transaction[3] which attempts to redeem T_A.

Now, to transfer 1Ƀ to another participant B, A can append to the blockchain the following transaction:

T_B
in: T_A
wit: $sig_{k_\mathsf{A}}(\mathsf{T}_\mathsf{B})$
out: $(\lambda x.\mathsf{versig}_{k_\mathsf{B}}(x), 1Ƀ)$

where the witness $sig_{k_\mathsf{A}}(\mathsf{T}_\mathsf{B})$ is A's signature on T_B (but for the wit field itself).

The ones shown above represent just the simplest cases of transactions. More in general, a Bitcoin transaction can collect bitcoins from many inputs, and split them between one or more outputs; further, it can use more complex scripts, and specify time constraints on when it can be appended to the blockchain.

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

- in contains the list of *inputs*. An input (T, i) refers to the i-th output of transaction T.
- wit contains the list of *witnesses*, of the same length as the list of inputs. For each input (T, i) in the in list, the witness at the same index must make the i-th output script of T evaluate to true.
- out contains the list of *outputs*. Each index refers to a pair $(\lambda \boldsymbol{z}.e, v)$, where the first component is a script, and the second is a currency value.
- absLock and relLock indicate absolute and relative time constraint on when the transaction can be added to the blockchain.

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 $n = 1$. We denote with $\widetilde{\mathsf{T}}^v_\mathsf{A}$ the *canonical* transaction, i.e. the transaction with a single output of the form $(\lambda\varsigma.\mathsf{versig}_{k_\mathsf{A}}(\varsigma), v Ƀ)$, and with all the other fields empty (denoted with $\bot$).

[3] Actually, the signature is not computed on the whole redeeming transaction, but only on a part of it, as shown in Section 2.3.

T_1	T_2	T_3
in: $\cdots$	in: $1 \mapsto (\mathsf{T}_1, 1)$	in: $1 \mapsto (\mathsf{T}_1, 2)$ $2 \mapsto (\mathsf{T}_2, 1)$
wit: $\cdots$	wit: $1 \mapsto \sigma_1$	wit: $1 \mapsto \sigma_2, \sigma_2'$ $2 \mapsto \sigma_3$
out: $1 \mapsto (\lambda x.\mathsf{versig}_k(x), v_1 ฿)$	out: $1 \mapsto (\lambda x.e_2, v_1 ฿)$	out: $1 \mapsto (\lambda x.e_3, (v_1 + v_2) ฿)$
$2 \mapsto (\lambda x, x'.e_1, v_2 ฿)$	relLock: $1 \mapsto t$	absLock: t'

Fig. 1: Three Bitcoin transactions.

Example 1. Consider the transactions in Figure 1. In T_1 there are two outputs: the first one transfers $v_1 ฿$ to any transaction T' which provides as witness a signature of T' with key k; the second output can transfer $v_2 ฿$ to a transaction whose witness satisfies the script e_1. The transaction T_2 tries to redeem $v_1 ฿$ from the output at index 1 of T_1, by providing the witness σ_1. Since $\mathsf{T}_2.\mathsf{relLock}(1) = t$, then T_2 can be appended only after at least t time units have passed since the transaction in $\mathsf{T}_2.\mathsf{in}(1)$ (i.e., T_1) appeared on the blockchain. In T_3, the input 1 refers to the output 2 of T_1, and the input 2 refers to the output 1 of T_2. The witness σ_2 and σ_2' are used to evaluate $\mathsf{T}_1.\mathsf{out}(2)$, replacing the occurrences of x and x' in e_1. Similarly, σ_3 is used to evaluate $\mathsf{T}_2.\mathsf{out}(1)$, replacing the occurrences of x in e_2. The transaction T_3 can be put on the blockchain only after time t'. □

2.2 Scripts

In Bitcoin, scripts are small programs written in a non-Turing equivalent language. Whoever provides a witness that makes the script evaluate to "true", can redeem the bitcoins retained in the associated (unspent) output. In the abstract model, scripts are terms of the form $\lambda \boldsymbol{z}.e$, where $\boldsymbol{z}$ is a sequence of variables occurring in e, and e is an expression with the following syntax:

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

Besides variables x, constants k, and basic arithmetic/logical operators, the other expression are peculiar: $|e|$ denotes the size, in bytes, of the evaluation of e; $\mathsf{H}(e)$ evaluates to the hash of e; $\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{e})$ evaluates to true iff the sequence of signatures $\boldsymbol{e}$ (say, of length m) is verified by using m out of the n keys in $\boldsymbol{k}$. For instance, the script $\lambda x.\mathsf{versig}_k(x)$ is satisfied if x is a signature on the redeeming transaction, verified with the key k. The expressions $\mathsf{absAfter}\ t : e$ and $\mathsf{relAfter}\ t : e$ define absolute and relative time constraints: they evaluate as e if the constraints are satisfied, otherwise they evaluate to false.

In Figure 2 we recap from [11] the semantics of script expressions. The function $[\![\cdot]\!]_{\mathsf{T},i,\rho}$ takes three parameters: T is the redeeming transaction, i is the index of the redeeming witness, and ρ is a map from variables to values. We use $\bot$ to represent the "failure" of the evaluation, H for a public hash function, and $size(n)$ for the size (in bytes) of an integer n. The function $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$ verifies a sequence of signatures $\boldsymbol{\sigma}$ against a sequence of keys $\boldsymbol{k}$ (see Section 2.3) All the

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

$$[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_{\mathsf{T},i,\rho} = \mathit{if}\ [\![e_0]\!]_{\mathsf{T},i,\rho}\ \mathit{then}\ [\![e_1]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ [\![e_2]\!]_{\mathsf{T},i,\rho}$$

$$[\![|e|]\!]_{\mathsf{T},i,\rho} = \mathit{size}([\![e]\!]_{\mathsf{T},i,\rho}) \quad [\![\mathsf{H}(e)]\!]_{\mathsf{T},i,\rho} = H([\![e]\!]_{\mathsf{T},i,\rho}) \quad [\![\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{e})]\!]_{\mathsf{T},i,\rho} = \mathsf{ver}_{\boldsymbol{k}}([\![\boldsymbol{e}]\!]_{\mathsf{T},i,\rho}, \mathsf{T}, i)$$

$$[\![\mathsf{absAfter}\ t : e]\!]_{\mathsf{T},i,\rho} = \mathit{if}\ \mathsf{T}.\mathsf{absLock} \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot$$

$$[\![\mathsf{relAfter}\ t : e]\!]_{\mathsf{T},i,\rho} = \mathit{if}\ \mathsf{T}.\mathsf{relLock}(i) \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot$$

Fig. 2: Semantics of script expressions.

semantic operators used in Figure 2 are *strict*, i.e. they evaluate to $\bot$ if some of their operands is $\bot$. We use syntactic sugar for expressions, e.g. *false* denotes $1 = 0$, *true* denotes $1 = 1$, while e and e' denotes $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ \mathit{false}$.

Example 2. Recall the transactions in Figure 1. Let e_1 (the script expression within $\mathsf{T}_1.\mathsf{out}(2)$) be defined as $e_1 = \mathsf{absAfter}\ t' : \mathsf{versig}_k(x)\ \mathsf{and}\ \mathsf{H}(x') = h$, for h and t' constants such that $\mathsf{T}_3.\mathsf{absLock} \geq t'$. Further, let σ_2 and σ_2' (the witnesses within $\mathsf{T}_3.\mathsf{wit}(1)$) be respectively $\mathsf{sig}_k(\mathsf{T}_3)$ and s, where $\mathsf{sig}_k(\mathsf{T}_3)$ is the signature of T_3 (excluding its witnesses) with key k, and s is a preimage of h, i.e. $h = H(s)$. Let $\rho = \{x \mapsto \mathsf{sig}_k(\mathsf{T}_3), x' \mapsto s\}$. To redeem $\mathsf{T}_1.\mathsf{out}(2)$ with the witness $\mathsf{T}_3.\mathsf{wit}(1)$, the script expression is evaluated as follows:

$$\begin{aligned}
&[\![\mathsf{absAfter}\ t' : \mathsf{versig}_k(x)\ \mathsf{and}\ \mathsf{H}(x') = h]\!]_{\mathsf{T}_3,1,\rho} \\
&= [\![\mathsf{versig}_k(x)\ \mathsf{and}\ \mathsf{H}(x') = h]\!]_{\mathsf{T}_3,1,\rho} && \text{as } \mathsf{T}_3.\mathsf{absLock} \geq t' \\
&= [\![\mathsf{versig}_k(x)]\!]_{\mathsf{T}_3,1,\rho} \ \wedge\ [\![\mathsf{H}(x') = h]\!]_{\mathsf{T}_3,1,\rho} \\
&= \mathsf{ver}_k(\rho(x), \mathsf{T}_3, 1) \ \wedge\ ([\![\mathsf{H}(x')]\!]_{\mathsf{T}_3,1,\rho} = [\![h]\!]_{\mathsf{T}_3,1,\rho}) \\
&= \mathsf{ver}_k(\mathsf{sig}_k(\mathsf{T}_3), \mathsf{T}_3, 1) \ \wedge\ (H(\rho(x')) = h) && \text{as } \rho(x) = \mathsf{sig}_k(\mathsf{T}_3) \\
&= \mathit{true} && \text{as } \rho(x') = s
\end{aligned}$$ □

2.3 Transaction signatures

The signatures verified with versig never apply to the whole transaction: the content of wit field is never signed, while the other fields can be excluded from the signature according to some predefined patterns. To sign parts of a transaction, we first erase the fields which we want to neglect in the signature. Technically, we set these fields to the "null" value $\bot$ using a *transaction substitution*.

A transaction substitution $\{\mathsf{f} \mapsto d\}$ replaces the content of field f with d. If the field is indexed (i.e., all fields but $\mathsf{absLock}$), we denote with $\{\mathsf{f}(i) \mapsto d\}$ the substitution of the i-th item in field f, and with $\{\mathsf{f}(\neq i) \mapsto d\}$ the substitution of all the items of field f *but* the i-th. For instance, to set *all* the elements of the wit field of T to $\bot$, we write $\mathsf{T}\{\mathsf{wit} \mapsto \bot\}$, and to additionally set the second input to $\bot$ we write $\mathsf{T}\{\mathsf{wit} \mapsto \bot\}\{\mathsf{in}(2) \mapsto \bot\}$.

In Bitcoin, there exists a fixed set of transaction substitutions. We represent them as *signature modifiers*, i.e. transaction substitutions which set to $\bot$ the

fields which will not be signed. Signatures never apply to the whole transaction: modifiers always discard the content of the wit, while they can keep all the inputs or only one, and all the outputs, or only one, or none. Modifiers also take a parameter i, which is instantiated to the index of the witness where the signature will be included. Below we only present two signature modifiers, since the others are not commonly used in Bitcoin smart contracts.

The modifier aa_i only sets the first witness to i, and the other witnesses to $\bot$ (so, *a*ll inputs and *a*ll outputs are signed). This ensures that a signature computed for being included in the witness at index i can not be used in any witness with index $j \neq i$:

$$aa_i(\mathsf{T}) \ = \mathsf{T}\{\mathsf{wit}(1) \mapsto i\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}$$

The modifier sa_i removes the witnesses, and all the inputs but the one at index i (so, a *s*ingle input and *a*ll outputs are signed). Differently from aa_i, this modifier discards the index i, so the signature can be included in any witness:

$$\begin{aligned} sa_i(\mathsf{T}) \ = \ aa_1(\mathsf{T}\{\mathsf{wit} \mapsto \bot\}\{\mathsf{in}(1) \mapsto \mathsf{T}.\mathsf{in}(i)\}\{\mathsf{in}(\neq 1) \mapsto \bot\} \\ \{\mathsf{relLock}(1) \mapsto \mathsf{T}.\mathsf{relLock}(i)\}\{\mathsf{relLock}(\neq 1) \mapsto \bot\}) \end{aligned}$$

Signatures carry information about which parts of the transaction are signed: formally, they are pairs $\sigma = (w, \mu)$, where μ is the modifier, and w is the signature on the transaction T modified with μ. We denote such signature as $\mathsf{sig}_k^{\mu,i}(\mathsf{T})$, where k is a key, and i is the index used by μ, if any. Verification of a signature σ for index i is denoted by $\mathsf{ver}_k(\sigma, \mathsf{T}, i)$. Formally:

$$\mathsf{sig}_k^{\mu,i}(\mathsf{T}) = (sig_k(\mu_i(\mathsf{T})), \mu) \qquad \mathsf{ver}_k(\sigma, \mathsf{T}, i) = ver_k(w, \mu_i(\mathsf{T})) \ \text{ if } \sigma = (w, \mu)$$

where *sig* and *ver* are, respectively, the signing function and the verification function of a digital signature scheme.

Multi-signature verification $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$ extends verification to the case where $\boldsymbol{\sigma}$ is a sequence of signatures and $\boldsymbol{k}$ is a sequence of keys. Intuitively, if $|\boldsymbol{\sigma}| = m$ and $|\boldsymbol{k}| = n$, it implements a m-of-n multi-signature scheme, evaluating to true if all the m signatures match (some of) the keys in $\boldsymbol{k}$. The actual definition also takes into account the order of signatures, as formalised in Definition 6 of [11].

2.4 Blockchain and consistency

Abstracting away from the fact that the actual Bitcoin blockchain is formed by blocks of transactions, here we represent a blockchain $\mathbf{B}$ as a sequence of pairs (T_i, t_i), where t_i is the time when T_i has been appended, and the values t_i are increasing. We say that the j-th output of the transaction T_i in the blockchain is *spent* (or, for brevity, that (T_i, j) is spent) if there exists some transaction $\mathsf{T}_{i'}$ in the blockchain (with $i' > i$) and some j' such that $\mathsf{T}_{i'}.\mathsf{in}(j') = (\mathsf{T}_i, j)$.

We now describe when a pair (T, t) can be appended to $\mathbf{B} = (\mathsf{T}_0, t_0) \cdots (\mathsf{T}_n, t_n)$. Following [11], we say that T is a *consistent update* of $\mathbf{B}$ at time t, in symbols $\mathbf{B} \rhd (\mathsf{T}, t)$, when the following conditions hold:

1. for each input i of T, if $\mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j)$ then:
   (a) T' corresponds to one of the transactions in $\mathbf{B}$;
   (b) (T', j) is *unspent* in $\mathbf{B}$;
   (c) the witness $\mathsf{T}.\mathsf{wit}(i)$ makes the script in $\mathsf{T}'.\mathsf{out}(j)$ evaluate to true;
2. the time constraints absLock and relLock in T are satisfied at time $t \geq t_n$;
3. the sum of the amounts of the inputs of T is greater or equal[4] to the sum of the amount of its outputs.

We assume that each transaction T_i in the blockchain is a consistent update of the sequence of past transactions $\mathsf{T}_0 \cdots \mathsf{T}_{i-1}$. The consistency of the blockchain is actually ensured by the Bitcoin consensus protocol.

Example 3. Recall the transactions in Figure 1. Assume a blockchain $\mathbf{B}$ whose last pair is (T_1, t_1) and $t_1 \geq t'$, while T_2 and T_3 are not in $\mathbf{B}$.

We verify that (T_2, t_2) is a consistent update of $\mathbf{B}$, assuming $t_2 = t_1 + t$ and that σ_1 is the signature of T_2 with (the private part of) key k. The only input of T_2 is $(\mathsf{T}_1, 1)$. Conditions 1a and 1b are satisfied, since $(\mathsf{T}_1, 1)$ is unspent in $\mathbf{B}$. Condition 1c holds because $\mathsf{versig}_k(\sigma_1)$ evaluates to true. Condition 2 holds: indeed the relative timelock in T_2 is satisfied because $t_2 - t_1 \geq t$. Condition 3 holds because the amount of the input of T_2, i.e. v_1Ð, is equal to the amount of its output. Note instead that (T_3, t_2) would *not* be a consistent update of $\mathbf{B}$, since it violates condition 1a on the second input.

Now, let $\mathbf{B}' = \mathbf{B}(\mathsf{T}_2, t_2)$. We verify that (T_3, t_3) is a consistent update of $\mathbf{B}'$, assuming $t_3 \geq t_2$, e_1 as in Example 2, and $e_2 = \mathsf{versig}_{k'}(x)$. Further, let $\sigma_2 = \mathsf{sig}_k(\mathsf{T}_3)$, let $\sigma_2' = s$, and $\sigma_3 = \mathsf{sig}_{k'}(\mathsf{T}_3)$. Conditions 1a and 1b hold, because T_1 and T_2 are in $\mathbf{B}'$, and the referred outputs are unspent. Condition 1c holds because the output scripts $\mathsf{T}_1.\mathsf{out}(2)$ and $\mathsf{T}_2.\mathsf{out}(1)$ against σ_2, σ_2' and σ_3 evaluate to true. Condition 2 is satisfied at $t_3 \geq t_2 \geq t_1 \geq t'$. Finally, condition 3 holds because the amount $(v_1 + v_2)$Ð in $\mathsf{T}_3.\mathsf{out}(1)$ is equal to the sum of the amounts in $\mathsf{T}_1.\mathsf{out}(2)$ and $\mathsf{T}_2.\mathsf{out}(1)$. □

3 Modelling Bitcoin contracts

In this section we introduce a formal model of the behavior of the participants in a contract, building upon the model of Bitcoin transactions in [11].

We start by formalising a simple language of expressions, which represent both the messages sent over the network, and the values used in internal computations made by the participants. Hereafter, we assume a set Var of *variables*, and we define the set Val of *values* comprising constants $k \in \mathbb{Z}$, signatures σ, scripts $\lambda \boldsymbol{z}.e$, transactions T, and currency values v.

[4] The difference between the amount of inputs and that of outputs is the *fee* paid to the miner who publishes the transaction.

$$[\![\nu]\!] = \nu \qquad [\![\mathsf{sig}_k^{\mu,i}(T)]\!] = \mathsf{sig}_k^{\mu,i}([\![T]\!]) \qquad [\![\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T, i)]\!] = \mathsf{ver}_{\boldsymbol{k}}([\![\boldsymbol{E}]\!], [\![T]\!], i)$$

$$[\![T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\}]\!] = [\![T]\!]\{\mathsf{f}(i) \mapsto [\![\boldsymbol{E}]\!]\} \qquad [\![(E, E')]\!] = ([\![E]\!], [\![E']\!])$$

$$[\![E \circ E']\!] = [\![E]\!] \circ [\![E']\!] \quad \text{for } \circ \in \{\ \mathsf{and}\ ,\ \mathsf{or}\ , +, \dots\} \qquad [\![\mathsf{not}\ E]\!] = \neg [\![E]\!]$$

$$[\![\boldsymbol{E}]\!] = [\![E_1]\!] \cdots [\![E_n]\!] \quad \text{if } \boldsymbol{E} = E_1 \cdots E_n$$

Fig. 3: Semantics of contract expressions.

Definition 1 (Contract expressions). *We define* contract expressions *through the following syntax:*

$E, T ::= \nu$		*value* ($\nu \in \mathsf{Val}$)
	x	*variable* ($x \in \mathsf{Var}$)
	$\mathsf{sig}_k^{\mu,i}(T)$	*signature* (μ *signature modifier*)
	$\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T, i)$	*(multi) signature verification*
	$T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\}$	*transaction field update*
	(E, E)	*pair*
	$E\ \mathsf{and}\ E \mid E\ \mathsf{or}\ E \mid \mathsf{not}\ E$	*logical expressions*
	$E + E \mid \cdots$	*arithmetic expressions*

where $\boldsymbol{E}$ denotes a finite sequence of expressions (i.e., $\boldsymbol{E} = E_1 \cdots E_n$). We define the function $[\![\cdot]\!]$ from (variable-free) contract expressions to values in Figure 3. As a notational shorthand, we omit the index i in sig *(resp.* versig*) when the signed (resp. verified) transactions have a single input.*

Intuitively, when T evaluates to a transaction T, the expression $T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\}$ represents the transaction obtained from T by substituting the field $\mathsf{f}(i)$ with the sequence of values obtained by evaluating $\boldsymbol{E}$. For instance, $\mathsf{T}\{\mathsf{wit}(1) \mapsto \sigma\}$ denotes the transaction obtained from T by replacing the witness at index 1 with the signature σ. Further, $\mathsf{sig}_k^{\mu,i}(T)$ evaluates to the signature of the transaction represented by T, and $\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T, i)$ represents the m-of-n multi-signature verification of the transaction represented by T. Both for the signing and verification, the parameter i represents the index where the signature will be used. We assume a simple type system (not specified here) that rules out ill-formed expressions, like e.g. $k\{\mathsf{wit}(1) \mapsto \mathsf{T}\}$.

We formalise the behaviour of a participant as an *endpoint protocol*, i.e. a process where the participant can perform the following actions: (i) send/receive messages to/from other participants; (ii) put a transaction on the ledger; (iii) wait until some transactions appear on the blockchain; (iv) do some internal computation. Note that the last kind of operation allows a participant to craft a transaction before putting it on the blockchain, e.g. setting the wit field to her signature, and later on adding the signature received from another participant.

Definition 2 (Endpoint protocols). *Assume a set of* participants *(named* A, B, C, $\ldots$*). We define* prefixes π*, and* protocols $P, Q, R, \ldots$ *as follows:*

π	$::= \mathsf{A}\,!\,\boldsymbol{E}$	*send messages to* A
	$\mid \mathsf{A}\,?\,\boldsymbol{x}$	*receive messages from* A
	$\mid \mathsf{put}\ T$	*append transaction* T *to the blockchain*
	$\mid \mathsf{ask}\ \boldsymbol{T}\ \mathsf{as}\ \boldsymbol{x}$	*wait until all transactions in* $\boldsymbol{T}$ *are on the blockchain*
	$\mid \mathsf{check}\ E$	*test condition*
P	$::= \sum_{i \in I} \pi_i \,.\, P_i$	*guarded choice (*I *finite set)*
	$\mid P \mid P$	*parallel composition*
	$\mid \mathrm{X}(\boldsymbol{E})$	*named process*

We assume that each name X *has a unique defining equation* $\mathrm{X}(\boldsymbol{x}) = P$ *where the free variables in* P *are included in* $\boldsymbol{x}$*. We use the following syntactic sugar:*

- $\tau \triangleq \mathsf{check}\ \mathit{true}$*, the* internal action*;*
- $\mathbf{0} \triangleq \sum_{\emptyset} P$*, the* terminated *protocol (as usual, we omit trailing* $\mathbf{0}$*s);*
- $\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleq \mathsf{check}\ E\ .\, P + \mathsf{check}\ \mathsf{not}\ E\ .\, Q$*;*
- $\pi_1.Q_1 + P \triangleq \sum_{i \in I \cup \{1\}} \pi_i.Q_i$*, provided that* $P = \sum_{i \in I} \pi_i.Q_i$ *and* $1 \notin I$*;*
- $\mathsf{let}\ x = E\ \mathsf{in}\ P \triangleq P\{^{E}/_{x}\}$*, i.e.* P *where* x *is replaced by* E*.*

The behaviour of protocols is defined in terms of a LTS between *systems*, i.e. the parallel composition of the protocols of all participants, and the blockchain.

Definition 3 (Semantics of protocols). *A* system S *is a term of the form* $\mathsf{A}_1[P_1] \mid \cdots \mid \mathsf{A}_n[P_n] \mid (\mathbf{B}, t)$*, where (i) all the* A_i *are distinct; (ii) there exists a single component* $(\mathbf{B}, t)$*, representing the current state of the blockchain* $\mathbf{B}$*, and the current time* t*; (iii) systems are up-to commutativity and associativity of* $\mid$*. We define the relation* $\rightarrow$ *between systems in Figure 4, where* $\mathit{match}_{\mathbf{B}}(\mathsf{T})$ *is the set of all the transactions in* $\mathbf{B}$ *that are equal to* T*, except for the witnesses. When writing* $S \mid S'$ *we intend that the conditions above are respected.*

Intuitively, a guarded choice $\sum_i \pi_i.P_i$ can behave as one of the branches P_i. A parallel composition $P \mid Q$ executes concurrently P and Q. All the rules (except the last two) specify how a protocol $(\pi.P + Q) \mid R$ evolves within a system. Rule [Com] models a message exchange between A and B: participant A sends messages $\boldsymbol{E}$, which are received by B on variables $\boldsymbol{x}$. Communication is synchronous, i.e. A is blocked until B is ready to receive. Rule [Check] allows the branch P of a sum to proceed if the condition represented by E is true. Rule [Put] allows A to append a transaction to the blockchain, provided that the update is consistent. Rule [Ask] allows the branch P of a sum to proceed only when the blockchain contains some transactions $\mathsf{T}'_1 \cdots \mathsf{T}'_n$ obtained by instantiating some $\bot$ fields in $\boldsymbol{T}$ (see Section 2). This form of pattern matching is crucial because the value of some fields (e.g., wit), may not be known at the time the protocol is written. When the ask prefix unblocks, the variables $\boldsymbol{x}$ in P are bound to

$$\mathsf{A}[\mathsf{B}\,!\,\boldsymbol{E}.\,P+R \mid Q] \mid \mathsf{B}[\mathsf{A}\,?\,\boldsymbol{x}.\,P'+R' \mid Q'] \mid S \to \mathsf{A}[P \mid Q] \mid \mathsf{B}[P'\{{}^{[\![\boldsymbol{E}]\!]}/_{\boldsymbol{x}}\} \mid Q'] \mid S \quad [\textsc{Com}]$$

$$\frac{[\![E]\!] = \mathit{true}}{\mathsf{A}[\mathsf{check}\ E\ .\ P+R \mid Q] \mid S \to \mathsf{A}[P \mid Q] \mid S}\ [\textsc{Check}]$$

$$\frac{[\![T]\!] = \mathsf{T} \qquad \mathbf{B} \rhd (\mathsf{T}, t)}{\mathsf{A}[\mathsf{put}\ T.\ P+R \mid Q] \mid S \mid (\mathbf{B}, t) \to \mathsf{A}[P \mid Q] \mid S \mid (\mathbf{B}(\mathsf{T}, t), t)}\ [\textsc{Put}]$$

$$\frac{[\![\boldsymbol{T}]\!] = \mathsf{T}_1 \cdots \mathsf{T}_n \quad \forall i \in 1..n : \mathit{match}_{\mathbf{B}}(\mathsf{T}_i) = \mathsf{T}'_i \neq \bot}{\mathsf{A}[\mathsf{ask}\ \boldsymbol{T}\ \mathsf{as}\ \boldsymbol{x}.\ P+R \mid Q] \mid S \mid (\mathbf{B}, t) \to \mathsf{A}[P\{{}^{\mathsf{T}'_1 \cdots \mathsf{T}'_n}/_{\boldsymbol{x}}\} \mid Q] \mid S \mid (\mathbf{B}, t)}\ [\textsc{Ask}]$$

$$\frac{\mathrm{X}(\boldsymbol{x}) = P \quad \mathsf{A}[P\{{}^{[\![\boldsymbol{E}]\!]}/_{\boldsymbol{x}}\} \mid Q] \mid S \to S'}{\mathsf{A}[\mathrm{X}(\boldsymbol{E}) \mid Q] \mid S \to S'}\ [\textsc{Def}] \qquad \frac{t' > 0}{S \mid (\mathbf{B}, t) \xrightarrow{t'} S' \mid (\mathbf{B}, t + t')}\ [\textsc{Delay}]$$

Fig. 4: Semantics of endpoint protocols.

T	T'_A	T'_B
in: $(\mathsf{T}_\mathsf{A}, 1)$	in: $(\mathsf{T}, 1)$	in: $(\mathsf{T}, 1)$
wit: $\bot$	wit: $\bot$	wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_\mathsf{A}\,k_\mathsf{B}}(\varsigma\varsigma'), 1\ddot{\mathrm{B}})$	out: $(\lambda\varsigma.\mathsf{versig}_{k_\mathsf{A}}(\varsigma), 1\ddot{\mathrm{B}})$	out: $(\lambda\varsigma.\mathsf{versig}_{k_\mathsf{B}}(\varsigma), 1\ddot{\mathrm{B}})$

Fig. 5: Transactions of the naïve escrow contract.

$\mathsf{T}'_1 \cdots \mathsf{T}'_n$, so making it possible to inspect their actual fields. Rule [Def] allows a named process $\mathrm{X}(\boldsymbol{E})$ to evolve as P, assuming a defining equation $\mathrm{X}(\boldsymbol{x}) = P$. The variables $\boldsymbol{x}$ in P are substituted with the results of the evaluation of $\boldsymbol{E}$. Such defining equations can be used to specify recursive behaviours. Finally, rule [Delay] allows time to pass[5].

Example 4 (Naïve escrow). A buyer A wants to buy an item from the seller B, but they do not trust each other. So, they would like to use a contract to ensure that B will get paid if and only if A gets her item. In a naïve attempt to realise this, they use the transactions in Figure 5, where we assume that $(\mathsf{T}_\mathsf{A}, 1)$ used in $\mathsf{T}.\mathsf{in}$, is a transaction output redeemable by A through her key k_A. The transaction T makes A deposit $1\ddot{\mathrm{B}}$, which can be redeemed by a transaction carrying the signatures of both A and B. The transactions T'_A and T'_B redeem T, transferring the money to A or B, respectively.

The protocols of A and B are, respectively, P_A and Q_B:

$$\begin{aligned} P_\mathsf{A} &= \mathsf{put}\ \mathsf{T}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T})\}.\ P' \\ P' &= \tau.\mathsf{B}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{B}) \ +\ \tau.\mathsf{B}\,?\,x.\ \mathsf{put}\ \mathsf{T}'_\mathsf{A}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{A})\,x\} \\ Q_\mathsf{B} &= \mathsf{ask}\ \mathsf{T}.\ \big(\tau.\mathsf{A}\,?\,x.\ \mathsf{put}\ \mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\} \ +\ \tau.\mathsf{A}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})\big) \end{aligned}$$

[5] To keep our presentation simple, we have not included time-constraining operators in endpoint protocols. In case one needs a finer-grained control of time, well-known techniques [39] exist to extend a process algebra like ours with these operators.

First, A adds her signature to T, and puts it on the blockchain. Then, she internally chooses whether to unblock the deposit for B or to request a refund. In the first case, A sends $\mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{B})$ to B. In the second case, she waits to receive the signature $\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})$ from B (saving it in the variable x); afterwards, she puts T'_A on the blockchain (after setting wit) to redeem the deposit. The seller B waits to see T on the blockchain. Then, he chooses either to receive the signature $\mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{B})$ from A (and then redeem the payment by putting T'_B on the blockchain), or to refund A, by sending his signature $\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})$.

This contract is not secure if either A or B are dishonest. On the one hand, a dishonest A can prevent B from redeeming the deposit, even if she had already received the item (to do that, it suffices not to send her signature, taking the rightmost branch in P'). On the other hand, a dishonest B can just avoid to send the item and the signature (taking the leftmost branch in Q_B): in this way, the deposit gets frozen. For instance, let $S = \mathsf{A}[P_\mathsf{A}] \mid \mathsf{B}[Q_\mathsf{B}] \mid (\mathbf{B}, t)$, where **B** contains T_A unredeemed. The scenario where A has never received the item, while B dishonestly attempts to receive the payment, is modelled as follows:

$$
\begin{aligned}
S &\to \mathsf{A}[P'] \mid \mathsf{B}[Q_\mathsf{B}] \mid (\mathbf{B}(\mathsf{T}, t), t) \\
&\to \mathsf{A}[P'] \mid \mathsf{B}[\tau.\mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\} \ +\ \tau.\mathsf{A}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})] \mid \cdots \\
&\to \mathsf{A}[\mathsf{B}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{A}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{A})\,x\}] \mid \mathsf{B}[\mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\}] \mid \cdots
\end{aligned}
$$

At this point the computation is stuck, because both A and B are waiting a message from the other participant. We will show in Section 4.3 how to design a secure escrow contract, with the intermediation of a trusted arbiter.

4 A survey of smart contracts on Bitcoin

We now present a comprehensive survey of smart contracts on Bitcoin, comprising those published in the academic literature, and those found online. To this aim we exploit the model of computation introduced in Section 3. Remarkably, all the following contracts can be implemented by only using so-called *standard* transactions[6], e.g. via the compilation technique in [11]. This is crucial, because non-standard transactions are currently discarded by the Bitcoin network.

4.1 Oracle

In many concrete scenarios one would like to make the execution of a contract depend on some real-world events, e.g. results of football matches for a betting contract, or feeds of flight delays for an insurance contract. However, the evaluation of Bitcoin scripts can not depend on the environment, so in these scenarios one has to resort to a trusted third-party, or *oracle* [2,19], who notifies real-world events by providing signatures on certain transactions.

For example, assume that A wants to transfer vɃ to B only if a certain event, notified by an oracle O, happens. To do that, A puts on the blockchain

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

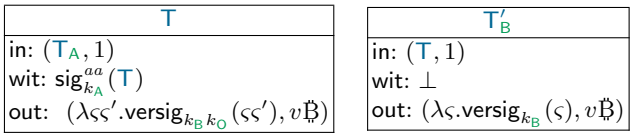

Fig. 6: Transactions of a contract relying on an oracle.

the transaction T in Figure 6, which can be redeemed by a transactions carrying the signatures of both B and O. Further, A instructs the oracle to provide his signature to B upon the occurrence of the expected event.

We model the behaviour of B as the following protocol:

$$P_{\mathsf{B}} \;=\; \mathsf{O}\,?\,x.\,\mathsf{put}\;\mathsf{T}'_{\mathsf{B}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{B}})\,x\}$$

Here, B waits to receive the signature $\mathsf{sig}^{aa}_{k_{\mathsf{O}}}(\mathsf{T}'_{\mathsf{B}})$ from O, then he puts T'_{B} on the blockchain (after setting its wit) to redeem T. In practice, oracles like the one needed in this contract are available as services in the Bitcoin ecosystem[7].

Notice that, in case the event certified by the oracle never happens, the v฿ within T are frozen forever. To avoid this situation, one can add a time constraint to the output script of T, e.g. as in the transaction T_{bond} in Figure 10.

4.2 Crowdfunding

Assume that the curator C of a crowdfunding campaign wants to fund a venture V by collecting v฿ from a set $\{\mathsf{A}_i\}_{i\in I}$ of investors. The investors want to be guaranteed that either the required amount v฿ is reached, or they will be able to redeem their funds. To this purpose, C can employ the following contract. She starts with a canonical transaction $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$ (with empty in field) which has a single output of v฿ to be redeemed by V. Intuitively, each A_i can invest money in the campaign by "filling in" the in field of the $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$ with a transaction output under their control. To do this, A_i sends to C a transaction output (T_i, j_i), together with the signature σ_i required to redeem it. We denote with $val(\mathsf{T}_i, j_i)$ the value of such output. Notice that, since the signature σ_i has been made on $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$, the only valid output is the one of v฿ to be redeemed by V. Upon the reception of the message from A_i, C updates $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$: the provided output is appended to the in field, and the signature is added to the corresponding wit field. If all the outputs (T_i, j_i) are distinct (and not redeemed) and the signatures are valid, when $\sum_i val(\mathsf{T}_i, j_i) \geq v$ the filled transaction $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$ can be put on the blockchain. If C collects $v' > v$฿, the difference $v' - v$ goes to the miners as transaction fee.

The endpoint protocol of the curator is defined as $\mathrm{X}(\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}, 1, 0)$, where:

$$\begin{aligned}\mathrm{X}(x, n, d) \;&=\; \mathsf{if}\; d < v\; \mathsf{then}\; P\; \mathsf{else}\; \mathsf{put}\; x\\ P \;&=\; \textstyle\sum_i \mathsf{A}_i\,?\,(y, j, \sigma).\,\mathrm{X}(x\{\mathsf{in}(n) \mapsto (y, j)\}\{\mathsf{wit}(n) \mapsto \sigma\}, n+1, d + val(y, j))\end{aligned}$$

[7] For instance, https://www.oraclize.it and https://www.smartcontract.com/

T	$\mathsf{T}'_{\mathsf{AB}}(z)$
in: $(\mathsf{T}_{\mathsf{A}}, 1)$	in: $(\mathsf{T}, 1)$
wit: $\bot$	wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_{\mathsf{A}}\,k_{\mathsf{B}}\,k_{\mathsf{C}}}(\varsigma\varsigma'), 1฿)$	out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{A}}}(\varsigma), z฿), 2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{B}}}(\varsigma), (1-z)฿)$

Fig. 7: Transactions of the escrow contract.

while the protocol of each investor A_i is the following:

$$P_{\mathsf{A}_i} = \mathsf{C}\,!\,(\mathsf{T}_i, j_i, \mathsf{sig}^{sa,1}_{k_{\mathsf{A}_i}}(\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}\{\mathsf{in}(1) \mapsto (\mathsf{T}_i, j_i)\}))$$

Note that the transactions sent by investors are not known *a priori*, so they cannot just create the final transaction and sign it. Instead, to allow C to complete the transaction $\widetilde{\mathsf{T}}^{v}_{\mathsf{V}}$ without invalidating the signatures, they compute them using the modifier sa_1. In this way, only a single input is signed, and when verifying the corresponding signature, the others are neglected.

4.3 Escrow

In Example 4 we have discussed a naïve escrow contract, which is secure only if both the buyer A and the seller B are honest (so making the contract pointless). Rather, one would like to guarantee that, even if either A or B (or both) are dishonest, exactly one them will be able to redeem the money: in case they disagree, a trusted participant C, who plays the role of arbiter, will decide who gets the money (possibly splitting the initial deposit in two parts) [1, 19].

The output script of the transaction T in Figure 7 is a *2-of-3* multi-signature schema. This means that T can be redeemed either with the signatures A and B (in case they agree), or with the signature of C (with key k_{C}) and the signature of A or that of B (in case they disagree). The transaction $\mathsf{T}'_{\mathsf{AB}}(z)$ in Figure 7 allows the arbiter to issue a *partial* refund of $z฿$ to A, and of $(1-z)฿$ to B. Instead, to issue a full refund to either A or B, the arbiter signs, respectively, the transactions $\mathsf{T}'_{\mathsf{A}} = \widetilde{\mathsf{T}}^{1฿}_{\mathsf{A}}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ or $\mathsf{T}'_{\mathsf{B}} = \widetilde{\mathsf{T}}^{1฿}_{\mathsf{B}}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ (not shown in the figure). The protocols of A and B are similar to those in Example 4, except for the part where they ask C for an arbitration:

$$\begin{aligned}
P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T})\}.\,(\tau.\mathsf{B}\,!\,\mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{B}}) \;+\; \tau.P') \\
P' &= \big(\mathsf{B}\,?\,x.\,(\mathsf{put}\ \mathsf{T}'_{\mathsf{A}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{A}})\,x\} + P'')\big) \;+\; P'' \\
P'' &= \mathsf{C}\,?\,(z,x).\,\big(\mathsf{check}\ z = 1\ .\ \mathsf{put}\ \mathsf{T}'_{\mathsf{A}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{A}})\,x\} \\
&\qquad + \mathsf{check}\ 0 < z < 1\ .\ \big(\mathsf{put}\ \mathsf{T}'_{\mathsf{AB}}(z)\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{AB}}(z))\,x\} + \tau.\mathbf{0}\big) \\
&\qquad + \mathsf{check}\ z = 0\ .\ \mathbf{0}\big)
\end{aligned}$$

In the summation within P_{A}, participant A internally chooses whether to send her signature to B (so allowing B to redeem $1฿$ via T'_{B}), or to proceed with P'. There, A waits to receive either B's signature (which allows A to redeem $1฿$

T_{AB}	T_{BC}
in: $(\mathsf{T}_{\mathsf{A}}, v_{\mathsf{C}})$	in: $(\mathsf{T}_{\mathsf{AB}}, 1)$
wit: $\bot$	wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_{\mathsf{A}}\,k_{\mathsf{B}}}(\varsigma\varsigma'), (v_{\mathsf{B}} + v_{\mathsf{C}})\ddot{\mathrm{B}})$	out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{B}}}(\varsigma), v_{\mathsf{B}}\ddot{\mathrm{B}})$, $2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{C}}}(\varsigma), v_{\mathsf{C}}\ddot{\mathrm{B}})$

Fig. 8: Transactions of the intermediated payment contract.

by putting T'_{A} on the blockchain), or a response from the arbiter, in the process P''. The three cases in the summation of check in P'' correspond, respectively, to the case where A gets a full refund ($z = 1$), a partial refund ($0 < z < 1$), or no refund at all ($z = 0$).

The protocol for B is dual to that of A:

$$
\begin{aligned}
Q_{\mathsf{B}} &= \mathsf{ask}\ \mathsf{T}.\,(\tau.\mathsf{A}\,!\,\mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{A}})\ +\ \tau.Q') \\
Q' &= \big(\mathsf{A}\,?\,x.\,(\mathsf{put}\ \mathsf{T}'_{\mathsf{B}}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{B}})\} + Q'')\big)\ +\ Q'' \\
Q'' &= \mathsf{C}\,?\,(z, x).\,\big(\mathsf{check}\ z = 0\ .\ \mathsf{put}\ \mathsf{T}'_{\mathsf{B}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{B}})\,x\} \\
&\qquad + \mathsf{check}\ 0 < z < 1\ .\ \big(\mathsf{put}\ \mathsf{T}'_{\mathsf{AB}}(z)\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{AB}}(z))\,x\} + \tau.\mathbf{0}\big) \\
&\qquad + \mathsf{check}\ z = 1\ .\ \mathbf{0}\big)
\end{aligned}
$$

If an arbitration is requested, C internally decides (through the τ actions) who between A and B can redeem the deposit in T, by sending its signature to one of the two participants, or decide for a partial refund of z and $1-z$ bitcoins, respectively, to A and B, by sending its signature on $\mathsf{T}'_{\mathsf{AB}}$ to both participants:

$$
\begin{aligned}
R_{\mathsf{C}} &= \mathsf{ask}\ \mathsf{T}.\,\big(\tau.\mathsf{A}\,!\,(1, \mathsf{sig}^{aa}_{k_{\mathsf{C}}}(\mathsf{T}'_{\mathsf{A}}))\ +\ \tau.\mathsf{B}\,!\,(1, \mathsf{sig}^{aa}_{k_{\mathsf{C}}}(\mathsf{T}'_{\mathsf{B}}))\ +\ \tau.R_{\mathsf{AB}}\big) \\
R_{\mathsf{AB}} &= \textstyle\sum_{0<z<1} \tau.\big(\mathsf{A}\,!\,(z, \mathsf{sig}^{aa}_{k_{\mathsf{C}}}(\mathsf{T}'_{\mathsf{AB}}(z)))\ |\ \mathsf{B}\,!\,(z, \mathsf{sig}^{aa}_{k_{\mathsf{C}}}(\mathsf{T}'_{\mathsf{AB}}(z)))\big)
\end{aligned}
$$

Note that, in the unlikely case where both A and B choose to send their signature to the other participant, the $1\ddot{\mathrm{B}}$ deposit becomes "frozen". In a more concrete version of this contract, a participant could keep listening for the signature, and attempt to redeem the deposit when (unexpectedly) receiving it.

4.4 Intermediated payment

Assume that A wants to send an indirect payment of $v_{\mathsf{C}}\ddot{\mathrm{B}}$ to C, routing it through an intermediary B who retains a fee of $v_{\mathsf{B}} < v_{\mathsf{C}}$ bitcoins. Since A does not trust B, she wants to use a contract to guarantee that: (i) if B is honest, then $v_{\mathsf{C}}\ddot{\mathrm{B}}$ are transferred to C; (ii) if B is *not* honest, then A does not lose money. The contract uses the transactions in Figure 8: T_{AB} transfers $(v_{\mathsf{B}} + v_{\mathsf{C}})\ddot{\mathrm{B}}$ from A to B, and T_{BC} splits the amount to B ($v_{\mathsf{B}}\ddot{\mathrm{B}}$) and to C ($v_{\mathsf{C}}\ddot{\mathrm{B}}$). We assume that $(\mathsf{T}_{\mathsf{A}}, 1)$ is a transaction output redeemable by A. The behaviour of A is as follows:

$$
\begin{aligned}
P_{\mathsf{A}} &= (\mathsf{B}\,?\,x.\ \mathsf{if}\ \mathsf{versig}_{k_{\mathsf{B}}}(x, \mathsf{T}_{\mathsf{BC}})\ \mathsf{then}\ P'\ \mathsf{else}\ \mathbf{0}) + \tau \\
P' &= \mathsf{put}\ \mathsf{T}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{AB}})\}.\ \mathsf{put}\ \mathsf{T}_{\mathsf{BC}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{BC}})\,x\}
\end{aligned}
$$

T_{com}
in: $(\mathsf{T}_\mathsf{A}, 1)$
wit: $\bot$
out: $(\lambda x \varsigma \varsigma'.(\mathsf{versig}_{k_\mathsf{A}}(\varsigma)$ and $\mathsf{H}(x) = h)$ or $\mathsf{versig}_{k_\mathsf{A}\,k_\mathsf{B}}(\varsigma\varsigma'), v\text{Ƀ})$

T_{open}
in: $(\mathsf{T}_{com}, 1)$
wit: $\bot$
out: $(\lambda\varsigma.\mathsf{versig}_{k_\mathsf{A}}(\varsigma), v\text{Ƀ})$

T_{pay}
in: $(\mathsf{T}_{com}, 1)$
wit: $\bot$
out: $(\lambda\varsigma.\mathsf{versig}_{k_\mathsf{B}}(\varsigma), v\text{Ƀ})$
relLock: t

Fig. 9: Transactions of the timed commitment.

Here, A receives from B his signature on T_{BC}, which makes it possible to pay C later on. The τ branch and the else branch ensure that A will correctly terminate also if B is dishonest (i.e., B does not send anything, or he sends an invalid signature). If A receives a valid signature, she puts T_{AB} on the blockchain, adding her signature to the wit field. Then, she also appends T_{BC}, adding to the wit field her signature and B's one. Since A takes care of publishing both transactions, the behaviour of B consists just in sending his signature on T_{BC}. Therefore, B's protocol can just be modelled as $Q_\mathsf{B} = \mathsf{A}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T}_{\mathsf{BC}})$.

This contract relies on SegWit. In Bitcoin without SegWit, the identifier of T_{AB} is affected by the instantiation of the wit field. So, when T_{AB} is put on the blockchain, the input in T_{BC} (which was computed before) does not point to it.

4.5 Timed commitment

Assume that A wants to choose a secret s, and reveal it after some time — while guaranteeing that the revealed value corresponds to the chosen secret (or paying a penalty otherwise). This can be obtained through a *timed commitment* [20], a protocol with applications e.g. in gambling games [25, 28, 42], where the secret contains the player move, and the delay in the revelation of the secret is intended to prevent other players from altering the outcome of the game. Here we formalise the version of the timed commitment protocol presented in [8].

Intuitively, A starts by exposing the hash of the secret, i.e. $h = H(s)$, and at the same time depositing some amount $v\text{Ƀ}$ in a transaction. The participant B has the guarantee that after t time units, he will either know the secret s, or he will be able to redeem $v\text{Ƀ}$.

The transactions of the protocol are shown in Figure 9, where we assume that $(\mathsf{T}_\mathsf{A}, 1)$ is a transaction output redeemable by A. The behaviour of A is modelled as the following protocol:

$$P_\mathsf{A} \;=\; \mathsf{put}\;\mathsf{T}_{com}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{com})\}.\,\mathsf{B}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{pay}).\,P'$$
$$P' \;=\; \tau\,.\,\mathsf{put}\;\mathsf{T}_{open}\{\mathsf{wit} \mapsto s\;\mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{open})\;\bot\}\;+\;\tau$$

Participant A starts by putting the transaction T_{com} on the blockchain. Note that within this transaction A is committing the hash of the chosen secret: indeed, h is encoded within the output script $\mathsf{T}_{com}.\mathsf{out}$. Then, A sends to B her signature on T_{pay}. Note that this transaction can be redeemed by B only when t time units have passed since T_{com} has been published on the blockchain, because

T_{bond}
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_{\mathsf{A}}\,k_{\mathsf{B}}}(\varsigma\varsigma')$ or relAfter $t : \mathsf{versig}_{k_{\mathsf{A}}}(\varsigma), k\ddot{\mathrm{B}})$

$\mathsf{T}_{pay}(v)$
in: $(\mathsf{T}_{bond}, 1)$
wit: $\bot$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{A}}}(\varsigma), (k-v)\ddot{\mathrm{B}})$ $2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{C}}}(\varsigma), v\ddot{\mathrm{B}})$

T_{ref}
in: $(\mathsf{T}_{bond}, 1)$
wit: $\bot$
out: $(\lambda\varsigma.\mathsf{versig}_{k_{\mathsf{A}}}(\varsigma), v\ddot{\mathrm{B}})$
relLock: t

Fig. 10: Transactions of the micropayment channel contract.

of the relative timelock declared in T_{pay}.relLock. After sending her signature on T_{pay}, A internally chooses whether to reveal the secret, or do nothing (via the τ actions). In the first case, A must put the transaction T_{open} on the blockchain. Since it redeems T_{com}, she needs to write in T_{open}.wit both the secret s and her signature, so making the former public.

A possible behaviour of the receiver B is the following:

$$Q_{\mathsf{B}} \;=\; \big(\mathsf{A}\,?\,x.\,\mathsf{if}\ \mathsf{versig}_{k_{\mathsf{A}}}(x, \mathsf{T}_{pay})\ \mathsf{then}\ Q\ \mathsf{else}\ \mathbf{0}\big) \;+\; \tau$$
$$Q \;=\; \mathsf{put}\ \mathsf{T}_{pay}\{\mathsf{wit} \mapsto \bot\ x\ \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}_{pay})\} + \mathsf{ask}\ \mathsf{T}_{open}\ \mathsf{as}\ o.\,Q'(get_{secret}(o))$$

In this protocol, B first receives from A (and saves in x) her signature on the transaction T_{pay}. Then, B checks if the signature is valid: if not, he aborts the protocol. Even if the signature is valid, B cannot put T_{pay} on the blockchain and redeem the deposit immediately, since the transaction has a timelock t. Note that B cannot change the timelock: indeed, doing so would invalidate A's signature on T_{pay}. If, after t time units, A has not published T_{open} yet, B can proceed to put T_{pay} on the blockchain, writing A's and his own signatures in the witness. Otherwise, B retrieves T_{open} from the blockchain, from which he can obtain the secret, and use it in Q'.

A variant of this contract, which implements the timeout in T_{com}.out, and does not require the signature exchange, is used in Section 4.7.

4.6 Micropayment channels

Assume that A wants to make a series of micropayments to B, e.g. a small fraction of Ƀ every few minutes. Doing so with one transaction per payment would result in conspicuous fees[8], so A and B use a micropayment channel contract [29]. A starts by depositing $k\ddot{\mathrm{B}}$; then, she signs a transaction that pays $v\ddot{\mathrm{B}}$ to B and $(k-v)\ddot{\mathrm{B}}$ back to herself, and she sends that transaction to B. Participant B can choose to publish that transaction immediately and redeem its payment, or to wait in case A sends another transaction with increased value. A can stop sending signatures at any time. If B redeems, then A can get back the remaining amount. If B does not cooperate, A can redeem all the amount after a timeout.

The protocol of A is the following (the transactions are in Figure 10). A publishes the transaction T_{bond}, depositing $k\ddot{\mathrm{B}}$ that can be spent with her signature

[8] https://bitinfocharts.com/comparison/bitcoin-transactionfees.html

and that of B, or with her signature alone, after time t. A can redeem the deposit by publishing the transaction T_{ref}. To pay for the service, A sends to B the amount v she is paying, and her signature on $\mathsf{T}_{pay}(v)$. Then, she can decide to increase v and recur, or to terminate.

$$\begin{aligned} P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}_{bond}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{bond})\}.\,(P(1) \mid \mathsf{put}\ \mathsf{T}_{ref}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{ref})\}) \\ P(v) &= \mathsf{B}\,!\,(v, \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{pay}(v)))).\,(\tau + \tau.P(v+1)) \end{aligned}$$

The participant B waits for T_{bond} to appear on the blockchain, then receives the first value v and A's signature σ. Then, B checks if σ is valid, otherwise he aborts the protocol. At this point, B waits for another pair (v', σ'), or, after a timeout, he redeems vɃ using $\mathsf{T}_{pay}(v)$.

$$\begin{aligned} Q_{\mathsf{B}} &= \mathsf{ask}\ \mathsf{T}_{bond}.\,\mathsf{A}\,?\,(v,\sigma).\,\mathsf{if}\ \mathsf{versig}_{k_{\mathsf{A}}}(\sigma, \mathsf{T}_{pay}(v))\ \mathsf{then}\ P'(v,\sigma)\ \mathsf{else}\ \tau \\ P'(v,\sigma) &= \tau.P_{pay}(v,\sigma)\ + \\ &\quad \mathsf{A}\,?\,(v',\sigma').\,\mathsf{if}\ v' > v\ \mathsf{and}\ \mathsf{versig}_{k_{\mathsf{A}}}(\sigma', \mathsf{T}_{pay}(v'))\ \mathsf{then}\ P'(v',\sigma')\ \mathsf{else}\ P'(v,\sigma) \\ P_{pay}(v,\sigma) &= \mathsf{put}\ \mathsf{T}_{pay}(v)\{\mathsf{wit} \mapsto \sigma\ \mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}_{pay}(v))\} \end{aligned}$$

Note that Q_{B} should redeem T_{pay} before the timeout expires, which is not modelled in Q_{B}. This could be obtained by enriching the calculus with time-constraining operators (see Footnote 5).

4.7 Fair lotteries

A multiparty lottery is a protocol where N players put their bets in a pot, and a winner — uniformly chosen among the players — redeems the whole pot. Various contracts for multiparty lotteries on Bitcoin have been proposed in [8, 9, 12, 14, 16, 36]. These contracts enjoy a *fairness* property, which roughly guarantees that: (i) each honest player will have (on average) a non-negative payoff, even in the presence of adversaries; (ii) when all the players are honest, the protocol behaves as an ideal lottery: one player wins the whole pot (with probability $1/N$), while all the others lose their bets (with probability $N-1/N$).

Here we illustrate the lottery in [8], for $N = 2$. Consider two players A and B who want to bet 1Ƀ each. Their protocol is composed of two phases. The first phase is a timed commitment (as in Section 4.5): each player chooses a secret (s_{A} and s_{B}) and commits its hash ($h_{\mathsf{A}} = H(s_{\mathsf{A}})$ and $h_{\mathsf{B}} = H(s_{\mathsf{B}})$). In doing that, both players put a deposit of 2Ƀ on the ledger, which is used to compensate the other player in case one chooses not to reveal the secret later on. In the second phase, the two bets are put on the ledger. After that, the players reveal their secrets, and redeem their deposits. Then, the secrets are used to compute the winner of the lottery in a fair manner. Finally, the winner redeems the bets.

The transactions needed for this lottery are displayed in Figure 11 (we only show A's transactions, as those of B are similar). The transactions for the commitment phase ($\mathsf{T}_{com}, \mathsf{T}_{open}, \mathsf{T}_{pay}$) are similar to those in Section 4.5: they only differ in the script of T_{com}.out, which now also checks that the length of the

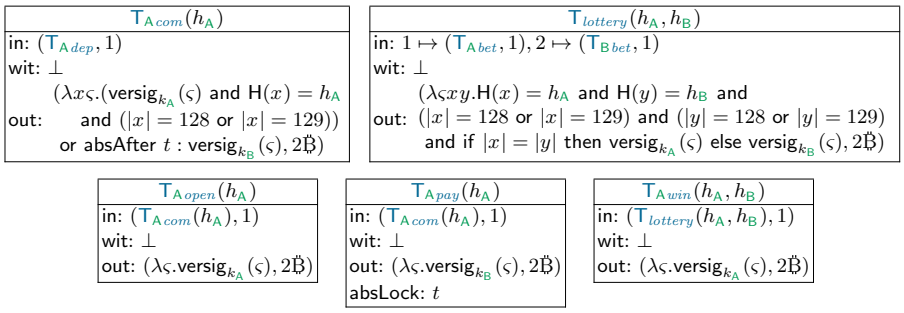

Fig. 11: Transactions of the fair lottery with deposit.

secret is either 128 or 129. This check forces the players to choose their secret so that it has one of these lengths, and reveal it (using T_{open}) before the absLock deadline, since otherwise they will lose their deposits (enabling T_{pay}).

The bets are put using $\mathsf{T}_{lottery}$, whose output script computes the winner using the secrets, which can then be revealed. For this, the secret lengths are compared: if equal, A wins, otherwise B wins. In this way, the lottery is equivalent to a coin toss. Note that, if a malicious player chooses a secret having another length than 128 or 129, the $\mathsf{T}_{lottery}$ transaction will become stuck, but its opponent will be compensated using the deposit.

The endpoint protocol P_A of player A follows (the one for B is similar):

$$
\begin{aligned}
P_\mathsf{A} &= \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,com}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{A}\,com})\}.\ (\mathsf{ask}\ \mathsf{T}_{\mathsf{B}\,com}\ \mathsf{as}\ y.\ P' \ +\ \tau.P_{open}) \\
P' &= \mathsf{let}\ h_\mathsf{B} = get_{hash}(y)\ \mathsf{in}\ \mathsf{if}\ h_\mathsf{B} \neq h_\mathsf{A}\ \mathsf{then}\ P_{pay} \mid P''\ \mathsf{else}\ P_{pay} \mid P_{open} \\
P'' &= \mathsf{B}\,?\,x.\ P''' \ +\ \tau.P_{open} \\
P''' &= \mathsf{let}\ \sigma = \mathsf{sig}^{aa,1}_{k_\mathsf{A}}(\mathsf{T}_{lottery}(h_\mathsf{A}, h_\mathsf{B}))\ \mathsf{in} \\
&\quad (\mathsf{put}\ \mathsf{T}_{lottery}(h_\mathsf{A}, h_\mathsf{B})\{\mathsf{wit}(1) \mapsto \sigma\}\{\mathsf{wit}(2) \mapsto x\}.\ (P_{open} \mid P_{win})) + \tau.P_{open} \\
P_{pay} &= \mathsf{put}\ \mathsf{T}_{\mathsf{B}\,pay}\{\mathsf{wit} \mapsto \bot\ \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{B}\,pay})\} \\
P_{open} &= \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,open}\{\mathsf{wit} \mapsto s_\mathsf{A}\ \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{A}\,open})\} \\
P_{win} &= \mathsf{ask}\ \mathsf{T}_{\mathsf{B}\,open}\ \mathsf{as}\ z.\ \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,win}(h_\mathsf{A}, h_\mathsf{B})\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{A}\,win}(h_\mathsf{A}, h_\mathsf{B}))\ s_\mathsf{A}\ get_{secret}(z)\}
\end{aligned}
$$

Player A starts by putting $\mathsf{T}_{\mathsf{A}\,com}$ on the blockchain, then she waits for B doing the same. If B does not cooperate, A can safely abort the protocol taking its $\tau.P_{open}$ branch, so redeeming her deposit with $\mathsf{T}_{\mathsf{A}\,open}$ (as usual, here with τ we are modelling a timeout). If B commits his secret, A executes P', extracting the hash h_B of B's secret, and checking whether it is distinct from h_A. If the hashes are found to be equal, A aborts the protocol using P_{open}. Otherwise, A runs $P'' \mid P_{pay}$. The P_{pay} component attempts to redeem B's deposit, as soon as the absLock deadline of $\mathsf{T}_{\mathsf{B}\,pay}$ expires, forcing B to timely reveal his secret. Instead, P'' proceeds with the lottery, asking B for his signature of $\mathsf{T}_{lottery}$. If B does not sign, A aborts using P_{open}. Then, A runs P''', finally putting the bets

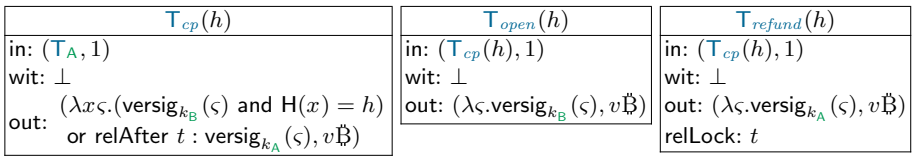

Fig. 12: Transactions of the contingent payment.

($\mathsf{T}_{lottery}$) on the ledger. If this is not possible (e.g., because one of the T_{bet} is already spent), A aborts using P_{open}. After $\mathsf{T}_{lottery}$ is on the ledger, A reveals her secret and redeems her deposit with P_{open}. In parallel, with P_{win} she waits for the secret of B to be revealed, and then attempts to redeem the pot ($\mathsf{T}_{\mathsf{A}win}$).

The fairness of this lottery has been established in [8]. This protocol can be generalised to $N > 2$ players [8,9] but in this case the deposit grows quadratically with N. The works [14,36] have proposed fair multiparty lotteries that require, respectively, zero and constant (≥ 0) deposit. More precisely, [36] devises two variants of the protocol: the first one only relies on SegWit, but requires each player to statically sign $O(2^N)$ transactions; the second variant reduces the number of signatures to $O(N^2)$, at the cost of introducing a custom opcode. Also the protocol in [14] assumes an extension of Bitcoin, i.e. the malleability of in fields, to obtain an ideal fair lottery with $O(N)$ signatures per player (see Section 5).

4.8 Contingent payments

Assume a participant A who wants to pay $v\ddot{\mathrm{B}}$ to receive a value s which makes a public predicate p true, where $p(s)$ can be verified efficiently. A seller B who knows such s is willing to reveal it to A, but only under the guarantee that he will be paid $v\ddot{\mathrm{B}}$. Similarly, the buyer wants to pay only if guaranteed to obtain s.

A naïve attempt to implement this contract in Bitcoin is the following: A creates a transaction T such that $\mathsf{T}.\mathsf{out}(\varsigma, x)$ evaluates to true if and only if $p(x)$ holds and ς is a signature of B. Hence, B can redeem $v\ddot{\mathrm{B}}$ from T by revealing s. In practice, though, this approach is arguably useful, since it requires coding p in the Bitcoin scripting language, whose expressiveness is quite limited.

More general contingent payment contracts can be obtained by exploiting zero-knowledge proofs [13,24,35]. In this setting, the seller generates a fresh key k, and sends to the buyer the encryption $e_s = E_k(s)$, together with the hash $h_k = H(k)$, and a zero-knowledge proof guaranteeing that such messages have the intended form. After verifying this proof, A is sure that B knows a preimage k' of h_k (by collision resistance, $k' = k$) such that $D_{k'}(e_s)$ satisfies the predicate p, and so she can buy the preimage k of h_k with the naïve protocol, so obtaining the solution s by decrypting e_s with k.

The transactions implementing this contract are displayed in Figure 12. The relAfter clause in T_{cp} allows A to redeem $v\ddot{\mathrm{B}}$ if no solution is provided by the

deadline t. The behaviour of the buyer A can be modelled as follows:

$$
\begin{aligned}
P_{\mathsf{A}} &= \mathsf{B}\,?\,(e_s, h_k, z).\, P \;+\; \tau \\
P &= \mathsf{if}\ \mathit{verify}(e_s, h_k, z)\ \mathsf{then\ put}\ \mathsf{T}_{cp}(h_k)\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{cp}(h_k))\}.\, P'\ \mathsf{else}\ \mathbf{0} \\
P' &= \mathsf{ask}\ \mathsf{T}_{open}(h_k)\ \mathsf{as}\ x.\, P''(D_{get_k(x)}(e_s)) \;+ \\
&\qquad \mathsf{put}\ \mathsf{T}_{refund}(h_k)\{\mathsf{wit} \mapsto \bot\ \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}_{refund}(h_k))\})
\end{aligned}
$$

Upon receiving e_s, h_k and the proof z[9] the buyer verifies z. If the verification succeeds, A puts $\mathsf{T}_{cp}(h_k)$ on the blockchain. Then, she waits for T_{open}, from which she can retrieve the key k, and so use the solution $D_{get_k(x)}(e_s)$ in P''. In this way, B can redeem vɃ. If B does not put T_{open}, after t time units A can get her deposit back through T_{refund}. The protocol of B is simple, so it is omitted.

5 Research challenges and perspectives

Extensions to Bitcoin. The formal model of smart contracts we have proposed is based on the current mechanisms of Bitcoin; indeed, this makes it possible to translate endpoint protocols into actual implementations interacting with the Bitcoin blockchain. However, constraining smart contracts to perfectly adhere to Bitcoin greatly reduces their expressiveness. Indeed, the Bitcoin scripting language features a very limited set of operations[10], and over the years many useful (and apparently harmless) opcodes have been disabled without a clear understanding of their alleged insecurity[11]. This is the case e.g., of bitwise logic operators, shift operators, integer multiplication, division and modulus.

For this reason some developers proposed to re-enable some disabled opcodes[12], and some works in the literature proposed extensions to the Bitcoin scripting language so to enhance the expressiveness of smart contracts.

A possible extension is *covenants* [37], a mechanism that allows an output script to constrain the structure of the redeeming transaction. This is obtained through a new opcode, called `CHECKOUTPUTVERIFY`, which checks if a given out of the redeeming transaction matches a specific pattern. Covenants are also studied in [41], where they are implemented using the opcode `CAT` (currently disabled) and a new opcode `CHECKSIGFROMSTACK` which verifies a signature against an arbitrary bitstring on the stack. In both works, covenants can also be recursive, e.g. a covenant can check if the redeeming transaction contains itself. Using recursive covenants allows to implement a state machine through a sequence of transactions that store its state.

[9] For simplicity, here we model the zero-knowledge proof as a single message. More concretely, it should be modelled as a sub-protocol.

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

[11] https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5141

[12] https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014356.html

Secure cash distribution with penalties [8, 16, 32] is a cryptographic primitive which allows a set of participants to make a deposit, and then provide inputs to a function whose evaluation determines how the deposits are distributed among the participants. This primitive guarantees that dishonest participants (who, e.g., abort the protocol after learning the value of the function) will pay a penalty to the honest participants. This primitive does not seem to be directly implementable in Bitcoin, but it becomes so by extending the scripting language with the opcode `CHECKSIGFROMSTACK` discussed above. Secure cash distribution with penalties can be instantiated to a variety of smart contracts, e.g. lotteries [8] poker [32], and contingent payments. The latter smart contract can also be obtained through the opcode `CHECKKEYPAIRVERIFY` in [24], which checks if the two top elements of the stack are a valid key pair.

Another new opcode, called `MULTIINPUT` [36] consumes from the stack a signature σ and a sequence of in values $(\mathsf{T}_1, j_1) \cdots (\mathsf{T}_n, j_n)$, with the following two effects: (i) it verifies the signature σ against the redeeming transaction T, neglecting $\mathsf{T}.\mathsf{in}$; (ii) it requires $\mathsf{T}.\mathsf{in}$ to be equal to some of the T_i. Exploiting this opcode, [36] devise a fair N-party lottery which requires zero deposit, and $O(N^2)$ off-chain signed transaction. The first one of these effects can be alternatively obtained by extending, instead of the scripting language, the signature modifiers. More specifically, [14] introduces a new signature modifier, which can set to $\bot$ *all* the inputs of a transaction (i.e., no input is signed). In this way they obtain a fair multi-party lottery with similar properties to the one in [36].

Another way improve the expressiveness of smart contracts is to replace the Bitcoin scripting language, e.g. with the one in [40]. This would also allow to establish bounds on the computational resources needed to run scripts.

Unfortunately, none of the proposed extensions has been yet included in the main branch of the Bitcoin Core client, and nothing suggests that they will be considered in the near future. Indeed, the development of Bitcoin is extremely conservative, as any change to its protocol requires an overwhelming consensus of the miners. So far, new opcodes can only be empirically assessed through the Elements alpha project[13], a testnet for experimenting new Bitcoin features. A significant research challenge would be that of formally proving that new opcodes do not introduce vulnerabilities, exploitable e.g. by Denial-of-Service attacks. For instance, unconstrained uses of the opcode `CAT` may cause an exponential space blow-up in the verification of transactions.

Formal methods for Bitcoin smart contracts. As witnessed in Section 4, designing secure smart contracts on Bitcoin is an error-prone task, similarly to designing secure cryptographic protocols. The reason lies in the fact that, to devise a secure contract, a designer has to anticipate any possible (mis-)behaviour of the other participants. The side effect is that endpoint protocols may be quite convoluted, as they must include compensations at all the points where something can go wrong. Therefore, tools to automate the analysis and verification of smart contracts may be of great help.

[13] https://elementsproject.org/elements/opcodes/

Recent works [7] propose to verify Bitcoin smart contracts by modelling the behaviour of participants as timed automata, and then using UPPAAL [15] to check properties against an attacker. This approach correctly captures the time constraints within the contracts. The downside is that encoding this UPPAAL model into an actual implementation with Bitcoin transactions is a complex task. Indeed, a designer without a deep knowledge of Bitcoin technicalities is likely to produce an UPPAAL model that can *not* be encoded in Bitcoin. A relevant research challenge is to study specification languages for Bitcoin contracts (like e.g. the one in Section 3), and techniques to *automatically* encode them in a model that can be verified by a model checker.

Remarkably, the verification of security properties of smart contracts requires to deal with non-trivial aspects, like temporal constraints and probabilities. This is the case, e.g., for the verification of fairness of lotteries (like e.g. the one discussed in Section 4.7); a further problem is that fairness must hold against any adversarial strategy. It is not clear whether in this case it is sufficient to consider a "most powerful" adversary, like e.g. in the symbolic Dolev-Yao model. In case a contract is not secure against arbitrary (PTIME) adversaries, one would like to verify that, at least, it is secure against *rational* ones [27], which is a relevant research issue. Additional issues arise when considering more concrete models of the Bitcoin blockchain, respect to the one in Section 2. This would require to model *forks*, i.e. the possibility that a recent transaction is removed from the blockchain. This could happen with rational (but dishonest) miners [33].

DSLs for smart contracts. As witnessed in Section 4, modelling Bitcoin smart contracts is complex and error-prone. A possible way to address this complexity is to devise high-level domain-specific languages (DSLs) for contracts, to be compiled in low-level protocols (e.g., the ones in Section 3). Indeed, the recent proliferation of non-Turing complete DSLs for smart contracts [18,22,26] suggests that this is an emerging research direction.

A first proposal of an high-level language implemented on top of Bitcoin is Typecoin [23]. This language allows to model the updates of a state machine as affine logic propositions. Users can "run" this machine by putting transactions on the Bitcoin blockchain. The security of the blockchain guarantees that only the legit updates of the machine can be triggered by users. A downside of this approach is that liveness is guaranteed only by assuming cooperation among the participants, i.e., a dishonest participant can make the others unable to complete an execution. Note instead that the smart contracts in Section 4 allow honest participants to terminate, regardless of the behaviours of the environment. In some cases, e.g. in the lottery in Section 4.7, abandoning the contract may even result in penalties (i.e., loss of the deposit paid upfront to stipulate the contract).

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. Bitcoin developer guide - escrow and arbitration. https://goo.gl/8XL5Fn
2. Bitcoin wiki - contracts - using external state. https://en.bitcoin.it/wiki/Contract#Example_4:_Using_external_state
3. Understanding the DAO attack (June 2016), http://www.coindesk.com/understanding-dao-hack-journalists/
4. Parity Wallet security alert (July 2017), https://paritytech.io/blog/security-alert.html
5. A Postmortem on the Parity Multi-Sig library self-destruct (November 2017), https://goo.gl/Kw3gXi
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)
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. pp. 7–22. Springer (2014)
8. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE Symposium on Security and Privacy. pp. 443–458 (2014)
9. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM 59(4), 76–84 (2016)
10. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Principles of Security and Trust (POST). LNCS, vol. 10204, pp. 164–186. Springer (2017), http://dx.doi.org/10.1007/978-3-662-54455-6_8
11. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Financial Cryptography and Data Security. LNCS, Springer (2018)
12. Back, A., Bentov, I.: Note on fair coin toss via Bitcoin. http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf (2013)
13. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS. LNCS, vol. 9879, pp. 261–280. Springer (2016)
14. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017)
15. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Formal methods for the design of real-time systems, LNCS, vol. 3185, pp. 200–236. Springer (2004)
16. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014)
17. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Beguelin, S.: Formal verification of smart contracts. In: PLAS (2016)
18. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: Secure derivative contracts for Ethereum. In: Financial Cryptography Workshops. LNCS, vol. 10323, pp. 453–467. Springer (2017)
19. BitFury group: Smart contracts on Bitcoin blockchain (2015), http://bitfury.com/content/5-white-papers-research/contracts-1.1.1.pdf
20. Boneh, D., Naor, M.: Timed commitments. In: CRYPTO. LNCS, vol. 1880, pp. 236–254. Springer (2000)
21. 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)

22. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: An introduction. http://r3cev.com/s/corda-introductory-whitepaper-final.pdf (2016)
23. Crary, K., Sullivan, M.J.: Peer-to-peer affine commitment using Bitcoin. In: ACM Conf. on Programming Language Design and Implementation. pp. 479–488 (2015)
24. Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: A fair protocol for data trading based on Bitcoin transactions. Future Generation Computer Systems (2017)
25. Delmolino, K., Arnett, M., Kosba, A.M.A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab (2016)
26. Frantz, C.K., Nowostawski, M.: From institutions to code: towards automated generation of smart contracts. In: eCAS Workshop (2016)
27. Garay, J.A., Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Rational protocol design: Cryptography against incentive-driven adversaries. In: FOCS. pp. 648–657 (2013)
28. Goldschlag, D.M., Stubblebine, S.G., Syverson, P.F.: Temporarily hidden bit commitment and lottery applications. Int. J. Inf. Sec. 9(1), 33–50 (2010)
29. Hearn, M.: Rapidly-adjusted (micro)payments to a pre-determined party (2013), bitcointalk.org
30. Hirai, Y.: Defining the Ethereum Virtual Machine for interactive theorem provers. In: Financial Cryptography Workshops. LNCS, vol. 10323, pp. 520–535. Springer (2017)
31. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014)
32. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015)
33. Liao, K., Katz, J.: Incentivizing blockchain forks via whale transactions. In: Financial Cryptography Workshops. LNCS, vol. 10323, pp. 264–279. Springer (2017)
34. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS (2016), http://eprint.iacr.org/2016/633
35. Maxwell, G.: The first successful zero-knowledge contingent payment. https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/ (2016)
36. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops. pp. 4–13 (2017)
37. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography Workshops. pp. 126–141. Springer (2016)
38. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
39. Nicollin, X., Sifakis, J.: An overview and synthesis on timed process algebras. In: CAV. LNCS, vol. 575, pp. 376–398 (1991)
40. O'Connor, R.: Simplicity: A new language for blockchains. In: PLAS (2017), http://arxiv.org/abs/1711.03028
41. O'Connor, R., Piekarska, M.: Enhancing Bitcoin transactions with covenants. In: Financial Cryptography Workshops (2017)
42. Syverson, P.F.: Weakly secret bit commitment: Applications to lotteries and fair exchange. In: IEEE CSFW. pp. 2–13 (1998)
43. 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