Fun with Bitcoin smart contracts

Massimo Bartoletti[1], Tiziana Cimoli[1], Roberto Zunino[2]

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

Abstract. Besides simple transfers of currency, Bitcoin also enables various forms of *smart contracts*, i.e. protocols where users interact within pre-agreed rules, which determine (possibly depending on the actual interaction) how currency is eventually distributed. This paper provides a gentle introduction to Bitcoin smart contracts, which we specify by abstracting from the underlying Bitcoin machinery. To this purpose we exploit BitML, a recent DSL for smart contracts executable on Bitcoin.

1 Introduction

Bitcoin and other cryptocurrencies [11,18] allow mutually distrusting parties to securely interact over a peer-to-peer network. Abstractly, Bitcoin can be seen as a decentralized state machine: the blockchain publicly records all the state transitions, and from the sequence of these transitions anyone can infer the state of the machine. The Bitcoin consensus mechanism guarantees that only the transitions which are consistent with the current state can be appended to the blockchain, and that previous transitions cannot be altered or removed.

The main use of Bitcoin so far is that of a cryptocurrency: state transitions record transfers of currency from one user to another one, and the state of the machine associates users to the amount of currency under their control. More in general, Bitcoin also enables various forms of *smart contracts*, i.e. protocols to distribute currency among users according to pre-agreed conditions [4,21]. A variety of protocols for lotteries [1,7,9,16], gambling games [15], contingent payments [6], payment channels [17], and other kinds of fair computations [2,14] witness the capabilities of Bitcoin as a machine for smart contracts.

In practice, the development of Bitcoin smart contracts has been hampered by the absence of convenient abstractions: indeed, existing descriptions of smart contracts require a thorough understanding of low-level features of Bitcoin, like e.g. transactions signatures and scripts.

In this paper we provide a gentle introduction to Bitcoin smart contracts by leveraging BitML [8], a recent high-level, process-algebraic language that compiles into Bitcoin transactions. The computational soundness of its compiler guarantees that the execution of the compiled contract is coherent with the semantics of the source BitML specification, even in the presence of adversaries. We start by specifying in BitML several smart contracts of growing complexity, intuitively describing their behaviour. Then, we show how to execute them on Bitcoin, by exploiting the BitML compiler.

2 Contracts

We illustrate Bitcoin smart contracts through a series of examples, without relying on any previous knowledge about Bitcoin. To this purpose we use BitML [8], a formalism which allows to express contracts in a process-algebraic fashion. In Section 3 we will show how to effectively execute these contracts on Bitcoin.

Contracts allow two or more participants (denoted as $\mathsf{A}, \mathsf{B}, \ldots$) to exchange their bitcoins (₿) according to the following workflow:

1. First, a participant broadcasts a *contract advertisement* $\{G\}C$. The component C is the actual contract, specifying the rules according to which the bitcoins can be transferred among participants. The component G is a set of *preconditions* to the execution of C. For instance, G can require participants to deposit some bitcoins, and to commit to some secrets.
2. If all the involved participants accept $\{G\}C$, satisfying its preconditions, the contract C becomes stipulated. Then, participants can interact, following the rules specified by C. According to the actual interaction, the final distribution of bitcoins among participants may vary.

2.1 Direct payment

Assume that A wants to give 1₿ to B through a contract. To this purpose, A must first declare that she owns 1₿, and that she agrees to transfer it under the control of the contract. This is represented by the following precondition:

$$G \;=\; \mathsf{A}:!\,1\text{₿} \tag{1}$$

while the actual contract is the following:

$$\mathit{Pay} \;=\; \texttt{withdraw}\,\mathsf{B} \tag{2}$$

We show below a possible computation of $\{G\}\mathit{Pay}$, using the semantics in [8] (which here we slightly simplify to ease the presentation). The configurations of the semantics are the parallel composition of terms of the following form (other terms needed for more advanced examples will be introduced later):

– $\{G\}C$, a contract advertisement;
– $\langle C, v\text{₿}\rangle$, a stipulated contract with a balance of v₿;
– $\langle \mathsf{A}, v\text{₿}\rangle_x$, a deposit of v₿ owned by A, and with unique name x;
– $\mathsf{A}[\chi]$, the *authorization* of A to perform some operation χ.

The initial configuration of our direct payment contract contains $\{G\}\mathit{Pay}$, and a deposit $\langle \mathsf{A}, 1\text{₿}\rangle_x$. The computation can then proceed as follows:

$$\begin{aligned}
\langle \mathsf{A}, 1\text{₿}\rangle_x \mid \{G\}\mathit{Pay} &\rightarrow \langle \mathsf{A}, 1\text{₿}\rangle_x \mid \{G\}\mathit{Pay} \mid \mathsf{A}[x \rhd \{G\}\mathit{Pay}] \\
&\rightarrow \langle \texttt{withdraw}\,\mathsf{B}, 1\text{₿}\rangle \\
&\rightarrow \langle \mathsf{B}, 1\text{₿}\rangle_y
\end{aligned}$$

At the first step, A authorizes the deposit x to be transferred to the contract, satisfying the precondition G. At the second step, Pay becomes stipulated, assimilating 1Ƀ from the deposit x (which is then removed from the configuration). At this point, the contract allows B to withdraw all its balance. When this happens, the contract becomes terminated (disappearing from the configuration), and a new deposit for B is added to the configuration.

2.2 Payment from multiple senders

In the previous contract, the initial deposit has been provided by a single participant, but more in general, a contract can gather money from multiple participants. For instance, assume A_1 and A_2 want to pay 1Ƀ each to B. We can perform this transfer atomically by using the following precondition:

$$G_2 \;=\; \mathsf{A}_1 \texttt{:!}\, 1Ƀ \;\mid\; \mathsf{A}_2 \texttt{:!}\, 1Ƀ \tag{3}$$

and the same contract Pay as in (2). Now, to stipulate the contract, both A_1 and A_2 must authorize to transfer their deposits to the contract:

$$\begin{aligned}
&\langle \mathsf{A}_1, 1Ƀ\rangle_x \mid \langle \mathsf{A}_2, 1Ƀ\rangle_y \mid \{G_2\}\mathit{Pay}\\
\to\; &\langle \mathsf{A}_1, 1Ƀ\rangle_x \mid \langle \mathsf{A}_2, 1Ƀ\rangle_y \mid \{G_2\}\mathit{Pay} \mid \mathsf{A}_1[x \rhd \{G_2\}\mathit{Pay}]\\
\to\; &\langle \mathsf{A}_1, 1Ƀ\rangle_x \mid \langle \mathsf{A}_2, 1Ƀ\rangle_y \mid \{G_2\}\mathit{Pay} \mid \mathsf{A}_1[x \rhd \{G_2\}\mathit{Pay}] \mid \mathsf{A}_2[y \rhd \{G_2\}\mathit{Pay}]\\
\to\; &\langle \mathit{Pay}, 1Ƀ\rangle \;\to\; \langle \mathsf{B}, 1Ƀ\rangle_z
\end{aligned}$$

Note that a similar behaviour could be obtained through the parallel composition of two advertisements, where A_1 and A_2 independently send 1Ƀ to B:

$$\{\mathsf{A}_1 \texttt{:!}\, 1Ƀ\}\; \texttt{withdraw}\; \mathsf{B} \;\mid\; \{\mathsf{A}_2 \texttt{:!}\, 1Ƀ\}\; \texttt{withdraw}\; \mathsf{B} \tag{4}$$

A remarkable difference between (3) and (4) is that, once stipulated, (3) guarantees that B will receive 2Ƀ; instead, in (4) there is no guarantee that if A_1 stipulates the contract, then also A_2 will do the same.

2.3 Procrastinating payments

Assume now that A wants to stipulate a contract where she commits herself to give 1Ƀ to B *after* a certain date d. For instance, this contract could represent a birthday present to be withdrawn only after the birthday date; or the paying of a rent to the landlord, to be withdrawn only after the 1st of the month. Using the same precondition in (1), A can use the following contract:

$$\mathit{PayAfter} \;=\; \texttt{after}\, d : \texttt{withdraw}\; \mathsf{B} \tag{5}$$

This contract locks the deposit until time d. After then, B can perform action `withdraw` B to redeem 1Ƀ from the contract, with no further time limitations. The computations must now record the passing of time: we do this by adding to the configuration a term $t = d_0$, meaning that the current global time is d_0. For instance, assuming that d_0 = 2018-04-01 and d = 2018-04-08, a possible computation of $\{G\}PayAfter$ is the following:

$$\begin{aligned} & \langle \mathsf{A}, 1\text{B} \rangle_x \mid \{G\}PayAfter \mid t = d_0 \to \cdots \to \langle PayAfter, 1\text{B} \rangle \mid t = d_0 \\ \xrightarrow{\text{7 days}} & \langle PayAfter, 1\text{B} \rangle \mid t = d \to \langle \mathsf{B}, 1\text{B} \rangle_y \mid t = d \end{aligned}$$

In the contract $PayAfter$, if B forgets to withdraw, the money remains within the contract. The following contract, instead, allows A to recover her money if B has not withdrawn within a given deadline $d' > d$:

$$PayOrRecover \;=\; \texttt{after}\, d : \texttt{withdraw}\ \mathsf{B} \;+\; \texttt{after}\, d' : \texttt{withdraw}\ \mathsf{A} \qquad (6)$$

where the precondition is the same as in (1). The contract allows two (mutually exclusive) behaviours: either A or B can withdraw 1B. Before the deadline d no one can withdraw; after d (but before d') only B can withdraw, while after the d' both `withdraw` actions are enabled, so the first one who performs their `withdraw` will get the money. This contract also models a "limited-time offer", which becomes unavailable after the deadline d'.

2.4 Authorizing payments

Assume that A is willing to pay 1B to B, but only if another participant O gives his authorization. With the precondition (1), we can use the following contract:

$$PayAuth \;=\; \mathsf{O} : \texttt{withdraw}\ \mathsf{B} \qquad (7)$$

A computation where O gives his authorization will then proceed as follows:

$$\begin{aligned} \langle \mathsf{A}, 1\text{B} \rangle_x \mid \{G\}PayAuth & \to \langle \mathsf{A}, 1\text{B} \rangle_x \mid \{G\}PayAuth \mid \mathsf{A}[x \rhd \{G\}PayAuth] \\ & \to \langle \mathsf{O} : \texttt{withdraw}\ \mathsf{B}, 1\text{B} \rangle \\ & \to \langle \mathsf{O} : \texttt{withdraw}\ \mathsf{B}, 1\text{B} \rangle \mid \mathsf{O}[\mathsf{O} : \texttt{withdraw}\ \mathsf{B}] \\ & \to \langle \mathsf{B}, 1\text{B} \rangle_y \end{aligned}$$

The semantics of contracts ensures that `withdraw` B can be performed only if the configuration contains a suitable authorization. In the computation above, this authorization is rendered by O[O : `withdraw` B], added at the third step[3].

We can play with authorizations and summations to construct more complex contracts. For instance, assume we want to design an *escrow* contract, which allows A to buy an item from B, authorizing the payment only after she gets the item. Further, B can authorize a full refund to A, in case there is some problem with the item. A naïve attempt to model this contract is the following:

$$NaiveEscrow \;=\; \mathsf{A} : \texttt{withdraw}\ \mathsf{B} \;+\; \mathsf{B} : \texttt{withdraw}\ \mathsf{A}$$

[3] To avoid ambiguities, the BitML semantics decorates contract terms with unique identifiers, referred to in authorization terms. Here we omit them for conciseness.

If both participants are honest, everything goes smoothly: when A receives the item, she authorizes the payment to B, otherwise B authorizes the refund. The problem with this contract is that, if neither A nor B give the authorization, the money in the contract is frozen. To cope with this issue, we can refine the escrow contract, by introducing a trusted arbiter O which resolves the dispute:

$$\mathit{OracleEscrow} \;=\; \mathit{NaiveEscrow} + \mathsf{O} : \texttt{withdraw}\ \mathsf{A} + \mathsf{O} : \texttt{withdraw}\ \mathsf{B}$$

The last two branches are used if neither A nor B give their authorizations: in this case, the arbiter chooses whether to authorize A or B to redeem the deposit. A variant of the escrow contract where O can issue a *partial* refund is in [8].

Another use case for authorizations is a bet, for instance on a football match. Two players A and B deposit 1₿ each, with precondition $\mathsf{A} : !\, 1₿ \mid \mathsf{B} : !\, 1₿$. The winner — determined by a trusted oracle O — can redeem the whole pot:

$$\mathsf{O} : \texttt{withdraw}\ \mathsf{A} + \mathsf{O} : \texttt{withdraw}\ \mathsf{B}$$

Note that a trusted oracle will only authorize the action corresponding the winner of the football match.

2.5 Splitting deposits

In all the previous examples, the deposit within the contract is transferred to a single participant. More in general, deposits can be split in many parts, to be transferred to different participants. For instance, assume that A wants her 1₿ deposit to be transferred in equal parts to B_1 and to B_2. Using the same precondition in (1), we can model this behaviour as follows:

$$\mathit{PaySplit} \;=\; \texttt{split}\ \big(0.5₿ \to \texttt{withdraw}\ \mathsf{B}_1 \mid 0.5₿ \to \texttt{withdraw}\ \mathsf{B}_2\big) \qquad (8)$$

The $\texttt{split}$ construct splits the contract in two or more parallel subcontracts, each with its own balance. Of course, the sum of their balances must be less than or equal to the deposit of the whole contract.

A possible computation of $\{G\}\mathit{PaySplit}$ is the following:

$$\begin{aligned}
\langle \mathsf{A}, 1₿\rangle_x \mid \{G\}\mathit{PaySplit} \;&\to \cdots \to \langle \mathit{PaySplit}, 1₿\rangle \\
&\to \langle \texttt{withdraw}\ \mathsf{B}_1, 0.5₿\rangle \mid \langle \texttt{withdraw}\ \mathsf{B}_2, 0.5₿\rangle \\
&\to \langle \mathsf{B}_1, 0.5₿\rangle_y \mid \langle \texttt{withdraw}\ \mathsf{B}_2, 0.5₿\rangle \\
&\to \langle \mathsf{B}_1, 0.5₿\rangle_y \mid \langle \mathsf{B}_2, 0.5₿\rangle_z
\end{aligned}$$

We can use $\texttt{split}$ together with the other primitives presented so far to craft more complex contracts. For instance, assume that A wants pay 0.9₿ to B, routing the payment through an intermediary I who can choose whether to authorize it (in this case retaining a 0.1₿ fee), or not. Since A does not trust I, she wants to use a contract to guarantee that: (i) if I authorizes the payment, then 0.9Ƀ are transferred to B; (ii) otherwise, A does not lose money. Using the same precondition in (1), we can model this behaviour as follows:

$$\mathsf{I} : \texttt{split}\ \big(0.1Ƀ \to \texttt{withdraw}\ \mathsf{I} \mid 0.9Ƀ \to \texttt{withdraw}\ \mathsf{B}\big) \ + \ \texttt{after}\ d : \texttt{withdraw}\ \mathsf{A}$$

The leftmost branch can only be taken if I authorizes the payment: in this case, I gets his fee, and B gets his payment. Instead, if I denies his authorization, then A can redeem her deposit after time d.

2.6 Volatile deposits

So far, we have seen participants using *persistent* deposits, that are assimilated by the contract upon stipulation. Besides these, participants can also use *volatile* deposits, which are *not* assimilated upon stipulation. For instance:

$$G_? \ = \ \mathsf{A} : ?\,0.5Ƀ\ \texttt{@}\,x \mid \mathsf{A} : !\ 0.5Ƀ$$

gives A the possibility of contributing 0.5Ƀ during the contract execution. However, A can choose instead to spend her volatile deposit outside the contract. The variable x is a handle to the volatile deposit, which can be used as follows:

$$Pay? \ = \ \texttt{put}\ x.\,\texttt{withdraw}\ \mathsf{B}$$

After stipulation, any participant can execute $\texttt{put}\ x$ to transfer the deposit x to the contract, provided that $\langle \mathsf{A}, 0.5Ƀ\rangle_x$ occurs in the configuration. Unlike the computation in Section 2.1, a computation of $\langle \mathsf{A}, 0.5Ƀ\rangle_x \mid \langle \mathsf{A}, 0.5Ƀ\rangle_y \mid \{G_?\}Pay?$ (even after stipulation) is not guaranteed to reach a configuration containing $\langle \mathsf{B}, 1Ƀ\rangle$. Indeed, since x is not paid upfront, there is no guarantee that x will be available when the contract demands it, as A can spend it for other purposes.

Volatile deposits can be exploited within more complex contracts, to handle situations where a participant wants to add some funds to the contract. For instance, assume a scenario where A_1 and A_2 want to give B 2Ƀ as a present, paying 1Ƀ each. However, A_2 is not sure *a priori* she will be able to pay, because she may need her 1Ƀ for more urgent purposes: in this case, A_1 is willing to pay an extra bitcoin. We can model this scenario as follows: A_1 puts 2Ƀ as a persistent deposit, while A_2 makes available a volatile deposit x of 1Ƀ:

$$\mathsf{A}_1 : !\ 2Ƀ \ \mid \ \mathsf{A}_2 : ?\ 1Ƀ\ \texttt{@}\,x$$

The contract is a choice between two branches:

$$\big(\texttt{put}\ x.\,\texttt{split}\ (2Ƀ \to \texttt{withdraw}\ \mathsf{B} \mid 1Ƀ \to \texttt{withdraw}\ \mathsf{A}_1)\big) + \texttt{after}\ d : \texttt{withdraw}\ \mathsf{B}$$

In the leftmost branch, A_2 puts 1Ƀ in the contract, and the balance is split between B (who takes 2Ƀ, as expected), and A_1 (who takes her extra deposit back). The rightmost branch is enabled after d, and it deals with the case where A_2 has not put her deposit by such deadline. In this case, B can redeem 2Ƀ, while A_2 loses the extra deposit. Note that, in both cases, B will receive 2Ƀ.

2.7 Revealing secrets

A useful feature of Bitcoin smart contracts is the possibility for a participant to choose a secret, and unblock some action only when the secret is revealed. Further, different actions can be enabled according to the length of the secret. Secrets must be declared in the contract precondition, as follows:

$$\mathsf{A} : \texttt{secret}\, a$$

We give the secret a *name*, here a, but we never denote the *value* of the secret itself. A basic contract which exploits this feature is the following:

$$\mathit{PaySecret} \;=\; \texttt{reveal}\, a \;\texttt{if}\; |a| > 1 \,.\, \texttt{withdraw}\; \mathsf{A} \tag{9}$$

This contract asks A to commit to a secret of length greater than one[4], and allows A to redeem 1₿ upon revealing the secret. Until then, the deposit is frozen.

In order to describe computations where participants commit to and reveal secrets, we extend configurations with two new kinds of terms:

- $\{\mathsf{A} : a\#N\}$, representing the fact that A has committed to a secret a. The length of a, which is secret as well, is determined by the integer N;
- $\mathsf{A} : a\#N$, representing the fact that A has revealed her secret a (hence, she has also revealed its length N).

Running $\{G \mid \mathsf{A} : \texttt{secret}\, a\}\mathit{PaySecret}$ with a secret of length 2 yields:

$$\begin{aligned} & \langle \mathsf{A}, 1₿ \rangle_x \mid \{\mathsf{A} : a\#2\} \mid \{G \mid \mathsf{A} : \texttt{secret}\, a\}\mathit{PaySecret} \;\rightarrow \cdots \\ \rightarrow\; & \{\mathsf{A} : a\#2\} \mid \langle \mathit{PaySecret}, 1₿ \rangle \\ \rightarrow\; & \mathsf{A} : a\#2 \mid \langle \mathit{PaySecret}, 1₿ \rangle \;\rightarrow\; \langle \texttt{withdraw}\; \mathsf{A}, 1₿ \rangle \;\rightarrow\; \langle \mathsf{A}, 1₿ \rangle_y \end{aligned}$$

The `reveal` primitive can be used to design more useful contracts than the one in (9). For instance, we show in (10) how to express a *timed commitment* contract [2, 10, 13, 20], using same the precondition as above. In this contract, A wants to choose a secret a, and reveal it before the deadline d; if A does not reveal the secret within d, B can redeem the 1₿ deposit as a compensation:

$$\mathit{TC} \;=\; \big(\texttt{reveal}\, a \,.\, \texttt{withdraw}\; \mathsf{A}\big) \;+\; \big(\texttt{after}\, d : \texttt{withdraw}\; \mathsf{B}\big) \tag{10}$$

Only A can choose the first branch, by revealing a. After that, anyone can further reduce the contract, and transfer 1₿ to A. Only after time d, if the `reveal` has not been performed, any participant can perform the `withdraw` in the second branch, which transfers 1₿ to B. Therefore, before the deadline A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). If no branch is taken by time d, a race condition occurs: in such case, the first one who fires the `withdraw` gets the money.

[4] After compiling to Bitcoin, the actual length of the secret will be increased by η, where η is a security parameter, large enough to avoid brute-force preimage attacks.

Using the precondition $\mathsf{A}\texttt{:!}\ 1\text{₿} \mid \mathsf{A}\texttt{:secret}\ a \mid \mathsf{B}\texttt{:!}\ 1\text{₿} \mid \mathsf{B}\texttt{:secret}\ b$, we can also model a *mutual* timed commitment as follows:

$$\begin{aligned}
\mathit{TC2} &= \texttt{reveal}\ a.\ C' + \texttt{after}\ d : \texttt{withdraw}\ \mathsf{B} \\
C' &= \texttt{reveal}\ b.\ C'' + \texttt{after}\ d' : \texttt{withdraw}\ \mathsf{A} && (d' > d) \\
C'' &= \texttt{split}\ \big(1\text{₿} \to \texttt{withdraw}\ \mathsf{A} \mid 1\text{₿} \to \texttt{withdraw}\ \mathsf{B}\big)
\end{aligned}$$

The contract $\mathit{TC2}$ can reduce to C' if A reveals a; otherwise (after d) B can redeem 2₿. If A reveals, then B can choose not to reveal. Doing so, however, B will lose his deposit, since, after d', A can withdraw the 2₿ deposited in the contract. Instead, if B reveals, the 2₿ are split between A and B. Any participant (either A or B) who behaves honestly is guaranteed to learn the other participant's secret, or to gain 1₿ as compensation — in this sense the protocol is fair. Note that d' must be sufficiently greater than d, to avoid the attack where A waits until the very last moment to reveal her secret, so making it difficult for B to respect the deadline.

2.8 Lotteries and other games

Now that we have introduced all the primitives of BitML, we can combine them to construct more advanced contracts. For instance, consider a multiparty lottery where n players put their bets in a pot, and a winner — fairly chosen among the players — redeems the whole pot.

We model a lottery similar to the one in [2,3], for two players A and B who bet 1₿ each. The contract preconditions are the following:

$$\mathsf{A}\texttt{:!}\ 3\text{₿} \mid \mathsf{A}\texttt{:secret}\ a \mid \mathsf{B}\texttt{:!}\ 3\text{₿} \mid \mathsf{B}\texttt{:secret}\ b \tag{11}$$

where the deposit of each player includes the 1₿ bet, plus a 2₿ collateral used as compensation in case of dishonest behaviour. The contract is the following:

$$\begin{aligned}
\texttt{split}\ \big(\ & 2\text{₿} \to \texttt{reveal}\ b\ \texttt{if}\ 0 \leq |b| \leq 1.\ \texttt{withdraw}\ \mathsf{B}\ +\ \texttt{after}\ d : \texttt{withdraw}\ \mathsf{A} \\
\mid\ & 2\text{₿} \to \texttt{reveal}\ a.\ \texttt{withdraw}\ \mathsf{A}\ +\ \texttt{after}\ d : \texttt{withdraw}\ \mathsf{B} \\
\mid\ & 2\text{₿} \to \texttt{reveal}\ ab\ \texttt{if}\ |a| = |b|.\ \texttt{withdraw}\ \mathsf{A} \\
& \qquad + \texttt{reveal}\ ab\ \texttt{if}\ |a| \neq |b|.\ \texttt{withdraw}\ \mathsf{B}\big)
\end{aligned}$$

The balance is split in three parts. Player B must reveal b by the deadline d; otherwise, A can redeem B's collateral (note that this is a timed commitment, similar to the one in (10)). Similarly, A must reveal a. To determine the winner we compare the lengths of the secrets, in the third part of the `split`. The winner is A if the secrets have the same length, otherwise it is B. Checking that b's length is either 0 or 1 is needed to achieve fairness: indeed, B can increase his probability to redeem 2₿ in the third part of the `split` by choosing a secret with length $N > 1$. However, doing so will make B lose his 2₿ deposit, so overall B's average payoff would be negative. A rational B would then choose a secret of length 0 or 1. Similarly, a rational A must choose a secret of length 0 or 1, otherwise she decreases her probability to be the winner. When both lengths are chosen in $\{0, 1\}$, both A and B can collect their collateral back, and they have a $1/2$ probability to win the lottery, provided that at least one of them chooses the length of the secret uniformly.

We also show a variant of the two-players lottery which requires no collateral, similarly to [7, 16]. The preconditions just require the 1₿ bets and the secrets, while the contract is the following, where $d' > d$:

$$\begin{aligned}
&\texttt{reveal}\, b\, \texttt{if}\, 0 \leq |b| \leq 1.\big(\, \texttt{reveal}\, a\, \texttt{if}\, |a| = |b|.\, \texttt{withdraw}\; \mathsf{A}\\
&\qquad\qquad\qquad\qquad\quad + \texttt{reveal}\, a\, \texttt{if}\, |a| \neq |b|.\, \texttt{withdraw}\; \mathsf{B}\\
&\qquad\qquad\qquad\qquad\quad + \texttt{after}\, d' : \texttt{withdraw}\; \mathsf{B}\,\big)\\
&+ \texttt{after}\, d : \texttt{withdraw}\; \mathsf{A}
\end{aligned}$$

Here, B must reveal first. If B does not reveal his secret by the deadline d, or the secret has not the expected length, then A can redeem 2₿. Otherwise, A in turn must reveal by the deadline d', or let B redeem 2₿. If both A and B reveal, then the winner is determined by comparing the lengths of their secrets. As before, the rational strategy for each player is to choose a secret length 0 or 1, and reveal it. This makes the lottery fair, even in the absence of a collateral.

Using similar insights, we can craft contracts for other games. For instance, consider *Rock-Paper-Scissors*, a two players hand game where both players choose simultaneously a hand-shape, and the winner is decided along with the following rules: rock beats scissors, scissors beats paper, and paper beats rock.

We model the game for two players A and B who bet 1₿ each, and represent their moves as secrets of length 0 (rock), 1 (paper), and 2 (scissors). We define the following boolean predicate to determine the winner:

$$w(N, M) \;=\; (N = 0 \wedge M = 2) \;\vee\; (N = 2 \wedge M = 1) \;\vee\; (N = 1 \wedge M = 0)$$

The contract preconditions are as in (11), while the contract is the following:

$$\begin{aligned}
&\texttt{split}\big(\\
&\;\; 2₿ \rightarrow \texttt{reveal}\, b\, \texttt{if}\, 0 \leq |b| \leq 2.\, \texttt{withdraw}\; \mathsf{B} \;+\; \texttt{after}\, d : \texttt{withdraw}\; \mathsf{A}\\
&\mid 2₿ \rightarrow \texttt{reveal}\, a\, \texttt{if}\, 0 \leq |a| \leq 2.\, \texttt{withdraw}\; \mathsf{A} \;+\; \texttt{after}\, d : \texttt{withdraw}\; \mathsf{B}\\
&\mid 2₿ \rightarrow \texttt{reveal}\, ab\, \texttt{if}\, w(|a|, |b|).\, \texttt{withdraw}\; \mathsf{A}\\
&\qquad + \texttt{reveal}\, ab\, \texttt{if}\, w(|b|, |a|).\, \texttt{withdraw}\; \mathsf{B}\\
&\qquad + \texttt{reveal}\, ab\, \texttt{if}\, |a| = |b|.\, \texttt{split}\; (1₿ \rightarrow \texttt{withdraw}\; \mathsf{A} \mid 1₿ \rightarrow \texttt{withdraw}\; \mathsf{B})\big)
\end{aligned}$$

The contract is split in three parts, each with a balance of 2₿: the first two parts allow the players to redeem the collaterals by revealing their secrets in time (similarly to the first version of the lottery), while the third one computes the winner. The winner is A if $w(|a|, |b|)$, and B if $w(|b|, |a|)$. If a and b have the same length (i.e., they represent the same move), then there is a tie, so the bets are given back to the two players. Notice that if a player chooses a secret of unexpected length, then it may happen that the 2₿ in the third part of the `split` remain frozen. However, in such case the dishonest player will pay a 2₿ penalty to the other one. A zero-collateral version of *Rock-Paper-Scissors* can be obtained similarly to the second version of the lottery.

3 From contracts to Bitcoin transactions

In this section we show how to execute on Bitcoin the contracts in Section 2. We start by providing some minimal background on Bitcoin. A transaction represents a transfer of bitcoins, and the sequence of all transactions is stored in a public, append-only data structure called *blockchain*. When a new transaction T is appended to the blockchain, it redeems bitcoins from one or more transactions already on the blockchain. For the aims of this paper we abstract from the fact that, in Bitcoin, there exist some transactions (so-called *coinbase*) which generate bitcoins from nothing, and that transactions are grouped into blocks.

The simplest Bitcoin transaction, which transfers 1₿ to participant A, can be represented as follows, using the notation in [5]:

T_A
in: T
wit: w
out: $(\lambda x.\mathsf{versig}_\mathsf{A}(x), 1₿)$

The transaction T_A is a record with three fields. The field in points to another transaction T, which must occur before T_A on the blockchain. The field out is a pair, whose first element is a boolean predicate (called *script* in the Bitcoin jargon), and the second element is the amount (1₿) deposited in T_A. To append T_A to the blockchain, T must contain at least 1₿. The script specifies the condition under which a subsequent transaction T' can redeem the 1₿ in T_A, transferring it to T'. In our case, the script requires a signature x of A on T'. To evaluate the script, the formal parameter x will be instantiated to the value of the wit field (called *witness*) of T'. In the previous figure, the witness w in T_A is the actual parameter used to evaluate the script in T, the transaction referred by T_A.in. If such evaluation yields true, then T_A can be appended to the blockchain, redeeming 1₿ from T. That sum is now under the control of A, since she is the only participant who can provide the needed witness in T'.

To execute a BitML contract, the involved users first translate it into a set of Bitcoin transactions, using the compiler in [8]. Then, they append one or more of these transactions to the Bitcoin blockchain. Intuitively, appending a transaction corresponds to a step of the contract execution, and so it may require users to perform the corresponding actions, like e.g. revealing a secret or providing an authorization. To compile contracts we will often exploit more advanced features of Bitcoin transactions than the above-mentioned ones, like e.g. that of collecting bitcoins from many inputs, and splitting them between many outputs. Further, we will often use more complex output scripts, and we will specify time constraints on when a transaction can be appended to the blockchain[5]. We will illustrate these features along with the examples where they are needed (see [5] for details).

[5] The BitML compiler always produces *standard* Bitcoin transactions, by exploiting the BALZaC tool (`https://github.com/balzac-lang/balzac`). This is crucial, since the Bitcoin network currently discards non-standard transactions.

3.1 Direct payment

Recall the contract advertisement $\{\mathsf{A}:!\,1₿\}\,\mathtt{withdraw}\,\mathsf{B}$ from Section 2.1. By exploiting the BitML compiler, A and B construct the following transactions:

$\mathsf{T}_{\mathit{init}}$
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda\varsigma_0\varsigma_1.\,\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1),\ 1₿)$

T'_{B}
in: $\mathsf{T}_{\mathit{init}}$
wit: $\mathsf{sig}_{\mathsf{A}}\ \mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\,\mathsf{versig}_{\mathsf{B}}(\varsigma),\ 1₿)$

where $\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1)$ is a shorthand for $\mathsf{versig}_{\mathsf{A}}(\varsigma_0) \wedge \mathsf{versig}_{\mathsf{B}}(\varsigma_1)$, and $\mathsf{sig}_{\mathsf{A}}$ represents A's signature on the enclosing transaction (similarly for $\mathsf{sig}_{\mathsf{B}}$).

In BitML, the stipulation of the contract starts with the following step:

$$\langle \mathsf{A}, 1₿\rangle_x \mid \{G\}\mathit{Pay} \ \rightarrow\ \langle \mathsf{A}, 1₿\rangle_x \mid \{G\}\mathit{Pay} \mid \mathsf{A}[x \rhd \{G\}\mathit{Pay}]$$

In Bitcoin, to perform this step the participants generate $\mathsf{T}_{\mathit{init}}$ and T'_{B} (which initially have an empty wit field), sign them, and exchange the signatures. After that, they insert the signatures in the wit fields as shown in the figure above. Crucially, the signature on $\mathsf{T}_{\mathit{init}}$ is broadcast by A only *after* B's signature has been received and verified. In this way, when $\mathsf{T}_{\mathit{init}}$ is put on the blockchain, starting the execution of the contract, A knows all the needed signatures to redeem it with T'_{B} later on. This guarantees that, after the contract starts executing, it can be run until completion. In BitML, A's signature on $\mathsf{T}_{\mathit{init}}$ is rendered as the authorization term $\mathsf{A}[x \rhd \{G\}\mathit{Pay}]$.

The second computation step in BitML is the following:

$$\langle \mathsf{A}, 1₿\rangle_x \mid \{G\}\mathit{Pay} \mid \mathsf{A}[x \rhd \{G\}\mathit{Pay}] \ \rightarrow\ \langle \mathtt{withdraw}\,\mathsf{B}, 1₿\rangle$$

In Bitcoin, this corresponds to appending $\mathsf{T}_{\mathit{init}}$ to the blockchain. This transaction redeems $1₿$ from T_{A} (displayed before at page 10) — the concrete counterpart of the BitML deposit $\langle \mathsf{A}, 1₿\rangle_x$. Note that, since both A and B know the witness of $\mathsf{T}_{\mathit{init}}$, any of them can append such transaction.

The last computation step in BitML is the following:

$$\langle \mathtt{withdraw}\,\mathsf{B}, 1₿\rangle \ \rightarrow\ \langle \mathsf{B}, 1₿\rangle_y$$

where y is a fresh name. In Bitcoin, this corresponds to appending to the blockchain the transaction T'_{B}, which redeems $1₿$ from $\mathsf{T}_{\mathit{init}}$. After that, $1₿$ is under B's control, since the script of T'_{B} only requires B's signature. The unspent transaction T'_{B} corresponds to the BitML deposit $\langle \mathsf{B}, 1₿\rangle_y$.

3.2 Payment from multiple senders

Recall $\{G_2\}\mathit{Pay} = \{\mathsf{A}_1:!\,1₿ \mid \mathsf{A}_2:!\,1₿\}\,\mathtt{withdraw}\,\mathsf{B}$ from Section 2.2. Assume that the deposits of A_1 and A_2 are provided by two transactions $\mathsf{T}_{\mathsf{A}_1}$ and $\mathsf{T}_{\mathsf{A}_2}$ similar to the transaction T_{A} at page 10 (but for the script). Although the initial deposits are more than one, we still use a single transaction T_{init} to gather them, by exploiting the fact that Bitcoin transactions can have multiple inputs. The compiler produces the following two transactions:

T_{init}
in: $0 \mapsto \mathsf{T}_{\mathsf{A}_1},\ 1 \mapsto \mathsf{T}_{\mathsf{A}_2}$
wit: $0 \mapsto \mathsf{sig}_{\mathsf{A}_1},\ 1 \mapsto \mathsf{sig}_{\mathsf{A}_2}$
out: $(\lambda\varsigma_1\varsigma_2\varsigma.\, \mathsf{versig}_{\mathsf{A}_1\mathsf{A}_2\mathsf{B}}(\varsigma_1\varsigma_2\varsigma),\ 2\text{₿})$

T'_{B}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}_1}\ \mathsf{sig}_{\mathsf{A}_2}\ \mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{B}}(\varsigma),\ 2\text{₿})$

Transaction T_{init} has two inputs: the one at index 0 points to $\mathsf{T}_{\mathsf{A}_1}$, while the other points to $\mathsf{T}_{\mathsf{A}_2}$. Consequently, it is possible to append T_{init} to the blockchain only if both $\mathsf{T}_{\mathsf{A}_1}$ and $\mathsf{T}_{\mathsf{A}_2}$ are still unredeemed on the blockchain. To this purpose, T_{init} needs to provide two witnesses, one for each input. In BitML, $\mathsf{T}_{\mathsf{A}_1}$ and $\mathsf{T}_{\mathsf{A}_2}$ are represented as deposits, say $\langle \mathsf{A}_1, 1\text{₿}\rangle_x$ and $\langle \mathsf{A}_2, 1\text{₿}\rangle_y$, and communicating the two signatures on T_{init} corresponds to providing the authorizations $\mathsf{A}_1[x \rhd \{G_2\}Pay]$ and $\mathsf{A}_2[y \rhd \{G_2\}Pay]$. As before, these signatures are exchanged only after all the other signatures have been exchanged and verified. Once the contract is stipulated, its execution proceeds as in Section 3.1.

3.3 Procrastinating payments

To deal with time constraints, we exploit the $\mathsf{absLock}$ field of the Bitcoin transaction: namely, setting $\mathsf{T}.\mathsf{absLock} = d$ prevents T from being appended to the blockchain before time d. For instance, recall *PayAfter* from (5). The BitML compiler produces the following two transactions:

T_{init}
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda\varsigma_0\varsigma_1.\, \mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1),\ 1\text{₿})$

T'_{B}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{B}}(\varsigma),\ 1\text{₿})$
absLock: d

In this way, even if after stipulation all the participants know all the witnesses, T'_{B} cannot be appended to the blockchain until time d, and as a consequence, B cannot use the 1₿ in T'_{B} before such date.

Recall now *PayOrRecover* from (6). The transactions obtained by compiling it are similar to the previous ones:

T_{init}
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda\varsigma_0\varsigma_1.\, \mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1), 1\text{₿})$

T'_{B}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{B}}(\varsigma),\ 1\text{₿})$
absLock: d

T'_{A}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{A}}(\varsigma),\ 1\text{₿})$
absLock: d'

The main difference with (5) is that now there are two transactions, T'_{B} and T'_{A}, that can redeem T_{init}. However, since T_{init} cannot be redeemed twice, only one of them can be appended to the blockchain: appending T'_{B} corresponds to executing the left branch of the choice, i.e. $\texttt{after}\, d : \texttt{withdraw}\ \mathsf{B}$, while T'_{A} corresponds to the right branch, i.e. $\texttt{after}\, d' : \texttt{withdraw}\ \mathsf{A}$.

3.4 Authorizing payments

As seen in the previous examples, to implement contract stipulation participants must exchange and verify their signatures on the Bitcoin transactions generated by the compiler. However, in case of contracts with authorizations, some signatures can be provided only during the execution of the contract, after stipulation.

For instance, compiling *PayAuth* in (7) results in the following transactions:

T_{init}
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda \varsigma_0 \varsigma_1 \varsigma_2 .\ \mathsf{versig}_{\mathsf{ABO}}(\varsigma_0 \varsigma_1 \varsigma_2),\ 1₿)$

T'_{B}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B}}\, [\mathsf{sig}_{\mathsf{O}}]$
out: $(\lambda \varsigma .\ \mathsf{versig}_{\mathsf{B}}(\varsigma),\ 1₿)$

where the square brackets around $\mathsf{sig}_{\mathsf{O}}$ in T'_{B} indicate that such signature does *not* need to be exchanged at stipulation time. Providing such signature at run time corresponds to the following computation step in BitML:

$$\langle \mathsf{O} : \texttt{withdraw}\ \mathsf{B}, 1₿ \rangle \rightarrow \langle \mathsf{O} : \texttt{withdraw}\ \mathsf{B}, 1₿ \rangle \mid \mathsf{O}[\mathsf{O} : \texttt{withdraw}\ \mathsf{B}]$$

Only after O's signature on T'_{B} is made public, it is possible to append such transaction to the blockchain, transferring 1₿ to B.

3.5 Splitting deposits

Bitcoin transactions can have multiple outputs: in this case, whoever redeems the transaction must specify *which* output it is redeeming. This feature is exploited to compile the `split` construct of BitML. For instance, compiling *PaySplit* from (8) produces the following transactions:

T_{init}
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda \varsigma_0 \varsigma_1 \varsigma_2 .\ \mathsf{versig}_{\mathsf{AB_1B_2}}(\varsigma_0 \varsigma_1 \varsigma_2),\ 1₿)$

T_{split}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B_1}}\, \mathsf{sig}_{\mathsf{B_2}}$
out: $0 \mapsto (\lambda \varsigma_0 \varsigma_1 \varsigma_2 .\ \mathsf{versig}_{\mathsf{AB_1B_2}}(\varsigma_0 \varsigma_1 \varsigma_2),\ 0.5₿)$
$1 \mapsto (\lambda \varsigma_0 \varsigma_1 \varsigma_2 .\ \mathsf{versig}_{\mathsf{AB_1B_2}}(\varsigma_0 \varsigma_1 \varsigma_2),\ 0.5₿)$

$\mathsf{T}'_{\mathsf{B_1}}$
in: $(\mathsf{T}_{split}, 0)$
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B_1}}\, \mathsf{sig}_{\mathsf{B_2}}$
out: $(\lambda \varsigma .\ \mathsf{versig}_{\mathsf{B_1}}(\varsigma),\ 0.5₿)$

$\mathsf{T}'_{\mathsf{B_2}}$
in: $(\mathsf{T}_{split}, 1)$
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B_1}}\, \mathsf{sig}_{\mathsf{B_2}}$
out: $(\lambda \varsigma .\ \mathsf{versig}_{\mathsf{B_2}}(\varsigma),\ 0.5₿)$

As usual, T_{init} gathers A's deposit and starts the contract. Then, appending T_{split} to the blockchain splits the contract balance between two different outputs, indexed with 0 and 1. In BitML, this would correspond to the computation step:

$$\langle \mathit{PaySplit}, 1₿ \rangle \rightarrow \langle \texttt{withdraw}\ \mathsf{B}_1, 0.5₿ \rangle \mid \langle \texttt{withdraw}\ \mathsf{B}_2, 0.5₿ \rangle$$

where the two contracts in the parallel composition can be executed independently (as usual in process calculi). Similarly, the two outputs of T_{split} can be independently redeemed by $\mathsf{T}'_{\mathsf{B}_1}$ and $\mathsf{T}'_{\mathsf{B}_2}$. The in field of these transactions specifies, besides the input transaction T_{split}, also the index of the output they want to redeem. Appending $\mathsf{T}'_{\mathsf{B}_1}$ corresponds, in BitML, to the step:

$$\langle \texttt{withdraw}\ \mathsf{B}_1, 0.5\text{₿}\rangle \mid \langle \texttt{withdraw}\ \mathsf{B}_2, 0.5\text{₿}\rangle \rightarrow \langle \mathsf{B}_1, 0.5\text{₿}\rangle_y \mid \langle \texttt{withdraw}\ \mathsf{B}_2, 0.5\text{₿}\rangle$$

3.6 Volatile deposits

Recall *Pay?* from Section 2.6, where A uses a volatile deposit x and a persistent one. Assume that the Bitcoin counterpart of x is a transaction T_x, from which A can redeem 0.5₿ by providing her own signature. The BitML compiler outputs:

T_{init}
in: T_A
wit: sig_A
out: $(\lambda\varsigma_0\varsigma_1.\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1), 0.5\text{₿})$

T_{put}
in: $0 \mapsto \mathsf{T}_{init}, 1 \mapsto \mathsf{T}_x$
wit: $0 \mapsto \mathsf{sig}_\mathsf{A}\mathsf{sig}_\mathsf{B}, 1 \mapsto \mathsf{sig}_\mathsf{A}$
out: $(\lambda\varsigma_0\varsigma_1.\,\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1),\ 1\text{₿})$

T'_A
in: T_{put}
wit: $\mathsf{sig}_\mathsf{A}\ \mathsf{sig}_\mathsf{B}$
out: $(\lambda\varsigma.\,\mathsf{versig}_\mathsf{A}(\varsigma),\ 1\text{₿})$

The transaction T_{init} gathers the persistent deposit, stored in T_A. The transaction T_{put} has two inputs: T_{init}, which can be redeemed with the signatures of both A and B, and T_x, which can be redeemed with A's signature only. Since all these signatures are exchanged before stipulation, any participant can append T_{put} to the blockchain — provided that T_x is still unspent. Instead, if T_x has been spent, the contract gets stuck, and the deposit within T_{init} is frozen.

3.7 Revealing secrets

Recall *PaySecret* from Section 2.7. In the stipulation phase, A commits to a secret (named a) and to its length N, by publishing the term $\{\mathsf{A} : a\#N\}$. In Bitcoin, this corresponds to choosing an actual bitstring s_a for the secret, and broadcasting its hash $h_a = H(s_a)$. To ensure that s_a cannot be recovered by brute force, even when N is small[6], we let the actual length of s_a be $\eta + N$, where η is a public security parameter, large enough (e.g., $\eta = 128$). In this way, the other participants cannot infer s_a (assuming H to be preimage resistant), nor its length. Further, A cannot later on reveal a different secret or a different length (assuming collision resistance). The BitML compiler generates the transactions:

T_{init}
in: T_A
wit: sig_A
out: $(\lambda\varsigma x.\mathsf{versig}_\mathsf{A}(\varsigma) \wedge H(x) = h_a \wedge \lvert x\rvert > \eta + 1,\ 1\text{₿})$

T_{reveal}
in: T_{init}
wit: $\mathsf{sig}_\mathsf{A}\ [s_a]$
out: $(\lambda\varsigma.\,\mathsf{versig}_\mathsf{A}(\varsigma),\ 1\text{₿})$

T'_A
in: T_{reveal}
wit: sig_A
out: $(\lambda\varsigma.\,\mathsf{versig}_\mathsf{A}(\varsigma),\ 1\text{₿})$

[6] The reason why BitML allows secrets to have small lengths is to make it easier to write some contracts, like e.g. those in Section 2.8.

Transaction T_{init} collects A's deposit, and its output script requires two witnesses: a signature ς of A on the redeeming transaction, and a bitstring x whose hash $H(x)$ is equal to h_a; further, x must be longer than $\eta + 1$ bits, to satisfy the condition $|a| > 1$ in the $\texttt{reveal} \cdots \texttt{if}$. These witnesses are provided by T_{reveal}, where the square brackets around s_a indicate that the secret can be provided *after* stipulation. Broadcasting the secret and appending T_{reveal} to the blockchain correspond to the following two BitML steps (which assume $N > 1$):

$$\{\mathsf{A} : a\#N\} \mid \langle \mathit{PaySecret}, 1\text{₿}\rangle \rightarrow \mathsf{A} : a\#N \mid \langle \mathit{PaySecret}, 1\text{₿}\rangle \rightarrow \langle \texttt{withdraw}\ \mathsf{A}, 1\text{₿}\rangle$$

Note that, once the transaction T_{reveal} is on the blockchain, everyone can read the secret in its wit field. After that, appending T'_{A} corresponds to the step:

$$\langle \texttt{withdraw}\ \mathsf{A}, 1\text{₿}\rangle \ \rightarrow\ \langle \mathsf{A}, 1\text{₿}\rangle_y$$

Recall now $\mathit{TC} = (\texttt{reveal}\, a.\texttt{withdraw}\ \mathsf{A}) + (\texttt{after}\, d : \texttt{withdraw}\ \mathsf{B})$, the timed commitment contract in (10). As before, we assume that A commits to a secret s_a by broadcasting its hash h_a. Further, we assume that A and B generate other two key pairs, $(k_{s\mathsf{A}}, k_{p\mathsf{A}})$ and $(k_{s\mathsf{B}}, k_{p\mathsf{B}})$, and share their public parts. The transactions obtained by the compiler are the following:

T_{init}
in: T_{A}
wit: $\mathsf{sig}_{\mathsf{A}}$
out: $(\lambda \varsigma_0 \varsigma_1 x.\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1) \vee (\mathsf{versig}_{k_{p\mathsf{A}} k_{p\mathsf{B}}}(\varsigma_0\varsigma_1) \wedge H(x) = h_a \wedge |x| \geq \eta), 1\text{₿})$

T'
in: T_{init}
wit: $\mathsf{sig}_{k_{s\mathsf{A}}}\, \mathsf{sig}_{k_{s\mathsf{B}}}\ [s_a]$
out: $(\lambda \varsigma_0 \varsigma_1.\mathsf{versig}_{\mathsf{AB}}(\varsigma_0\varsigma_1), 1\text{₿})$

T'_{A}
in: T'
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{A}}(\varsigma), 1\text{₿})$

T'_{B}
in: T_{init}
wit: $\mathsf{sig}_{\mathsf{A}}\, \mathsf{sig}_{\mathsf{B}}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathsf{B}}(\varsigma), 1\text{₿})$
absLock: d

The transaction T_{init} can be redeemed in two ways, according to the two clauses in the disjunction of its output script: either with the signatures $\mathsf{sig}_{\mathsf{A}}$ and $\mathsf{sig}_{\mathsf{B}}$, or with the signatures $\mathsf{sig}_{k_{s\mathsf{A}}}$ and $\mathsf{sig}_{k_{s\mathsf{B}}}$ *and* the secret value s_a.

In the first case one can use the transaction T'_{B}, which however can be appended only after time d, because of the time constraint specified in its absLock field. Appending T'_{B} corresponds to the following step in BitML (where $d' \geq d$):

$$\langle \mathit{TC}, 1\text{₿}\rangle \mid t = d' \ \rightarrow\ \langle \mathsf{B}, 1\text{₿}\rangle_y \mid t = d'$$

In the second case, one can use the transaction T', by filling its wit field with the secret s_a revealed by A. Doing this corresponds to the following computation steps in BitML (which can be performed at any time):

$$\{\mathsf{A} : a\#N\} \mid \langle \mathit{TC}, 1\text{₿}\rangle \ \rightarrow\ \mathsf{A} : a\#N \mid \langle \mathit{TC}, 1\text{₿}\rangle \ \rightarrow\ \langle \texttt{withdraw}\ \mathsf{A}, 1\text{₿}\rangle$$

After that, anyone can append the transaction T'_{A} to the blockchain to transfer 1₿ under A's control. Once T' is on the blockchain, it will be no longer possible to append T'_{B}, since both transactions want to redeem T_{init}.

4 Related work and conclusions

We have illustrated Bitcoin smart contracts from a programming languages perspective, by exploiting the BitML calculus [8]. Although BitML can express many of the smart contracts appeared in literature [4], there exist some contracts which can be executed on Bitcoin but are not expressible in BitML. This is the case e.g. of *contingent payment* contracts, where a participant A promises to pay B for a value x satisfying a predicate chosen by A (e.g., x is a prime factor of a given large number). Contingent payments can be implemented in Bitcoin similarly to timed commitment contracts: A pays a deposit, which is taken by B after revealing a preimage of $H(x)$ which satisfies the predicate. An off-chain protocol [6] (which exploits zero-knowledge proofs) is used to guarantee that $H(x)$ is indeed the hash of a value x satisfying the predicate (note that, in the Bitcoin scripting language, one can only check trivial predicates, like e.g. equality). Another kind of contracts which are not expressible in BitML are those for which one cannot pre-determine a *finite* set of transactions, or of signatures, before executing the contract. This is the case, e.g., of crowdfunding contracts [4], where participants invest some money until a given threshold is reached. Here, we do not statically know neither the number of participants, nor their identities, so it is not possible to statically produce (and pre-sign) a set of transactions, as required by BitML. To the best of our knowledge, the existence of negative results on the expressiveness of Bitcoin contracts is still an open question

Only a few other languages for Bitcoin contracts have been proposed so far. TypeCoin [12] is an high-level language which allows to model the updates of a state machine as affine logic propositions. Users can "run" this machine by putting transactions on the blockchain, with the guarantee that only the legit updates can be performed. A downside of [12] is that liveness is guaranteed only by assuming cooperative participants, i.e., a dishonest participant can make the others unable to complete an execution. Note instead that in BitML, honest participants can always make a contract progress, regardless of the behaviour of the environment. Cooperation is incentivized by punishing misbehaviour with penalties, like e.g. in the lottery of Section 2.8. The other works we are aware of, IVY[7], BALZaC[8] and Simplicity [19], replace the Bitcoin scripting language with more high-level languages, through which they simplify writing the transactions needed in a smart contract (e.g., by providing static checks to determine if a transaction can redeem another one, etc.). Compared to these approaches, BitML completely abstracts from Bitcoin transactions, in this way allowing for more elegant specifications of contracts (compare e.g. the lotteries in Section 2.8 with those in [3, 7, 16]), and paving the way towards automatic verification.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia projects "Sardcoin" and "Smart collaborative engineering".

[7] `https://ivy-lang.org/bitcoin`
[8] `https://blockchain.unica.it/balzac/`

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 S & P. pp. 443–458 (2014)
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM 59(4), 76–84 (2016)
4. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: SoK: unraveling Bitcoin smart contracts. In: Principles of Security and Trust (POST) (2018)
5. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Financial Cryptography and Data Security (2018)
6. 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)
7. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Financial Cryptography Workshops (2017), also in IACR Cryptology ePrint Archive 955/2016
8. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. Cryptology ePrint Archive, Report 2018/122 (2018), https://eprint.iacr.org/2018/122
9. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014)
10. Boneh, D., Naor, M.: Timed commitments. In: CRYPTO. LNCS, vol. 1880, pp. 236–254. Springer (2000)
11. 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)
12. 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)
13. 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)
14. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014)
15. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015)
16. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops. pp. 4–13 (2017)
17. Miller, A., Bentov, I., Kumaresan, R., McCorry, P.: Sprites: Payment channels that go faster than lightning. CoRR abs/1702.05812 (2017), http://arxiv.org/abs/1702.05812
18. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
19. O'Connor, R.: Simplicity: A new language for blockchains. In: PLAS (2017), http://arxiv.org/abs/1711.03028
20. Syverson, P.F.: Weakly secret bit commitment: Applications to lotteries and fair exchange. In: IEEE CSFW. pp. 2–13 (1998)
21. 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