Renegotiation and Recursion in Bitcoin Contracts

Massimo Bartoletti, Maurizio Murgia, Roberto Zunino

Renegotiation and recursion in Bitcoin contracts

Massimo Bartoletti[1], Maurizio Murgia[2], and Roberto Zunino[2]

[1] University of Cagliari, Italy
[2] University of Trento, Italy

Abstract. BitML is a process calculus to express smart contracts that can be run on Bitcoin. One of its current limitations is that, once a contract has been stipulated, the participants cannot renegotiate its terms: this prevents expressing common financial contracts, where funds have to be added by participants at run-time. In this paper, we extend BitML with a new primitive for contract renegotiation. At the same time, the new primitive can be used to write recursive contracts, which was not possible in the original BitML. We show that, despite the increased expressiveness, it is still possible to execute BitML on standard Bitcoin, preserving the security guarantees of BitML.

1 Introduction

Smart contracts — computer protocols that regulate the exchange of assets in trustless environments — have become popular with the growth of interest in blockchain technologies. Mainstream blockchain platforms like Ethereum, Libra, and Cardano, feature expressive high-level languages for programming smart contracts. This flexibility has a drawback in that it may open the door to attacks that steal or tamper with the assets controlled by vulnerable contracts [4,22].

An alternative approach, pursued first by Bitcoin and more recently also by Algorand, is to sacrifice the expressiveness of smart contracts to reduce the attack surface. For instance, Bitcoin has a minimal language for transaction redeem scripts, containing only a limited set of logic, arithmetic, and cryptographic operations. Despite the limited expressiveness of these scripts, it is possible to encode a variety of smart contracts (like gambling games, escrow services, crowdfunding systems, etc.) by suitably chaining transactions [1,2,3,5,8,10,13,18,19,20,21,23]. The common trait of these works is that they render contracts as cryptographic protocols, where participants can exchange/sign messages, read the blockchain, and append transactions. Verifying the correctness of these protocols is hard, since it requires to reason in a computational model, where participants can manipulate arbitrary bitstrings, only being constrained to use PPTIME algorithms.

Departing from this approach, BitML [11] allows to write Bitcoin contracts in a high-level, process-algebraic language. BitML features a compiler that translates contracts into sets of standard Bitcoin transactions, and a sound and complete verification technique of relevant trace properties [12]. The computational soundness of the 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. Although BitML can express many of the Bitcoin contracts presented in the literature, it is not "Bitcoin-complete", i.e. there exist contracts that can be executed on Bitcoin, but are not expressible in BitML [6].

For instance, consider a zero-coupon bond [17], where an investor A pays 1฿ upfront to a bank B, and receives back 2฿ after a maturity date (say, year 2030). We can express this contract in BitML as follows. First, as a precondition to the stipulation of the contract, we require both A and B to provide a deposit: A's deposit is 1฿, while B's deposit is 2฿. In BitML, we write this precondition as:

$$\mathsf{A}: 1฿ \,@\, x_1 \mid \mathsf{B}: 2฿ \,@\, x_2$$

where x_1 and x_2 are the identifiers of transactions containing the required amount of bitcoins (฿). Under this precondition, we can specify the zero-coupon bond contract *ZCB* as follows:

$$\mathit{ZCB} \;=\; \texttt{split}\,\big(1฿ \to \texttt{withdraw}\;\mathsf{B} \;\mid\; 2฿ \to \texttt{after}\,2030 : \texttt{withdraw}\;\mathsf{A}\big)$$

Upon stipulation, all the deposits required in the preconditions pass under the control of *ZCB*, and can no longer be spent by A and B. The contract splits these funds in two parts: 1฿, that can be withdrawn by B at any moment, and 2฿, that can be withdrawn by A after the maturity date.

Although *ZCB* correctly implements the functionality of zero-coupon bounds, it is quite impractical: for the whole period from the stipulation to the maturity date, 2฿ are frozen within the contract, and cannot be used by the bank in any way. Although this is a desirable feature for the investor, since it guarantees that he will receive 2฿ even if the bank fails, it is quite undesirable for the bank. In the real world, the bank would be free to use its own funds, together with those of investors, to make further financial transactions through which to repay the investments. The risk that the bank fails is mitigated by external mechanisms, like insurances or government intervention.

In this paper we propose an extension of BitML that overcomes this issue. The idea is to allow the contract participants to *renegotiate* it after stipulation, in a controlled way. Renegotiation makes it possible to inject in the contract new funds, that were not specified in the original precondition. We can use this feature to solve the issue with the *ZCB* contract. The new precondition is $\mathsf{A}: 1฿ \,@\, x_1$, i.e. we only require A's deposit. The revised contract is:

$$\begin{aligned} \mathit{ZCB2} \;&=\; \texttt{split}\,\big(1฿ \to \texttt{withdraw}\;\mathsf{B} \;\mid\; 0฿ \to * : \texttt{rngt}\;\mathtt{X}\langle\rangle\big) \\ \mathtt{X}\langle\rangle \;&=\; \{\mathsf{B}: 2฿ \,@\, d\}\;\texttt{after}\,2030 : \texttt{withdraw}\;\mathsf{A} \end{aligned}$$

As before, the bank can withdraw 1฿ at any moment after stipulation. In the second part of the `split`, the participants renegotiate the contract: if they both agree, 0฿ pass under the control of the contract $\mathtt{X}\langle\rangle$. The precondition of $\mathtt{X}\langle\rangle$ requires the bank to provide 2฿ in a fresh deposit; upon renegotiation, A can withdraw 2฿ after the maturity date. The crucial difference with *ZCB* is that the deposit variable d is instantiated at *renegotiation* time, unlike x, which must be fixed at *stipulation* time.

The revised contract *ZCB2* solves the problem of *ZCB*, in that it no longer freezes $2\ddot{\mathrm{B}}$ for the whole duration of the bond: the bank could choose to renegotiate the contract, paying $2\ddot{\mathrm{B}}$, just before the maturity date. This flexibility comes at a cost, since A loses the guarantee to eventually receive $2\ddot{\mathrm{B}}$. To address this issue we need to add, as in the real world, an external mechanism. More specifically, we assume an insurance company I that, for an annual premium of $p\ddot{\mathrm{B}}$ paid by the bank, covers a face amount of $f\ddot{\mathrm{B}}$ (with $2 > f > 10p$):

$$\mathsf{A}\colon 1\ddot{\mathrm{B}} \mathbin{@} x_1 \mid \mathsf{B}\colon p\ddot{\mathrm{B}} \mathbin{@} x_2 \mid \mathsf{I}\colon f\ddot{\mathrm{B}} \mathbin{@} x_3$$

We revise the bond contract as follows:

$$\begin{aligned}
\mathit{ZCB3} \;&=\; \mathtt{split}\,\big(1\ddot{\mathrm{B}} \to \mathtt{withdraw}\;\mathsf{B}\\
&\qquad\quad \mid p\ddot{\mathrm{B}} \to \mathtt{withdraw}\;\mathsf{I}\\
&\qquad\quad \mid f\ddot{\mathrm{B}} \to * : \mathtt{rngt}\;\mathtt{X}\langle 1\rangle + \mathtt{after}\,2021 : \mathtt{withdraw}\;\mathsf{A}\big)\\
\mathtt{X}\langle n \in 1..9\rangle \;&=\; \{\mathsf{B}\colon p\ddot{\mathrm{B}} \mathbin{@} d\}\\
&\quad\; \mathtt{split}\,\big(p\ddot{\mathrm{B}} \to \mathtt{withdraw}\;\mathsf{I}\\
&\qquad\quad \mid f\ddot{\mathrm{B}} \to * : \mathtt{rngt}\;\mathtt{X}\langle n+1\rangle + \mathtt{after}\,(2021+n) : \mathtt{withdraw}\;\mathsf{A}\big)\\
\mathtt{X}\langle 10\rangle \;&=\; \{\mathsf{B}\colon 2\ddot{\mathrm{B}} \mathbin{@} d\}\\
&\quad\; \mathtt{split}\,\big(f\ddot{\mathrm{B}} \to \mathtt{withdraw}\;\mathsf{I}\\
&\qquad\quad \mid 2\ddot{\mathrm{B}} \to \mathtt{after}\,2030 : \mathtt{withdraw}\;\mathsf{A}\big)
\end{aligned}$$

The contract starts by transferring $1\ddot{\mathrm{B}}$ to the bank, and the first year of the premium to the insurer. The remaining $f\ddot{\mathrm{B}}$ are transferred to the renegotiated contract $\mathtt{X}\langle 1\rangle$, or, if the renegotiation is not completed by 2021, to the investor.

We remark that the pattern $D + \mathtt{after}\,t : D'$, where D requires some authorizations but D' does not, is rather common in BitML, as it ensures that the contract can proceed even if the authorizations are not provided. Indeed, in such case an honest participant is enough to execute D' after time t. By suitably exploiting this pattern, it is possible to guarantee that a BitML contract enjoys liveness, by just assuming that at least one participant is honest.

The contracts $\mathtt{X}\langle n\rangle$, for $n \in 1..9$, allow the insurer to receive the annual premium until 2030: if the bank does not renegotiate the contract for the following year (paying the corresponding premium), then the investor can redeem the face amount of $f\ddot{\mathrm{B}}$. Finally, the contract $\mathtt{X}\langle 10\rangle$ can be triggered if the bank deposits the $2\ddot{\mathrm{B}}$: when this happens, the face amount is given back to the insurer, and the investor can redeem $2\ddot{\mathrm{B}}$ after the maturity date.

Compared to *ZCB2*, the contract *ZCB3* offers more protection to the investor. To see why, we must evaluate A's payoff for all the possible behaviours of the other participants. If B and I are both honest, then A will redeem $2\ddot{\mathrm{B}}$, as in the ideal contract *ZCB*. Instead, if either B or I do not accept to renegotiate some $\mathtt{X}\langle n\rangle$, then A can redeem $f\ddot{\mathrm{B}}$ as a partial compensation (unlike in *ZCB2*, where A just loses $1\ddot{\mathrm{B}}$). In the real world, A could use this compensation to cover the legal fee to sue the bank in court; also, I could e.g. increase the premium

for future interactions with B. By further refining the contract, we could model these real-world mechanisms as oracles, which sanction dishonest participants according to the evidence collected in the blockchain and in messages broadcast by participants. For instance, if B and I accept the renegotiation $\mathtt{X}\langle n\rangle$ but A does not, then the oracle would be able to detect A's dishonesty by inspecting the authorizations broadcast in year $2021+n$. The sanction could consist e.g. in blacklisting A, so to prevent her from buying other bonds from B.

Contributions. We summarise our main contributions as follows:

- We extend BitML with the renegotiation primitive $* : \mathtt{rngt}\ \mathtt{X}\langle\rangle$, suitably adapting the language syntax and semantics. The new primitive increases the expressiveness of BitML: besides allowing participants to provide new deposits and secrets at run-time, it also allows for *unbounded* recursion.
- We extend the BitML compiler to the new primitive, making it possible to execute renegotiations on Bitcoin. We accordingly extend the computational soundness result in [11], guaranteeing that the BitML semantics is coherent with the actual Bitcoin executions, also in the presence of adversaries.
- We exploit renegotiation to design a new gambling game where two players repeatedly flip coins, and whoever wins twice in a row takes the pot (a form of unbounded recursion). We prove the game to be fair.
- We introduce alternative renegotiation primitives, which allow participants to choose some parameters (e.g. the amounts to be deposited) at renegotiation time, and to change the set of participants involved in the renegotiated contract. We show that both primitives can be executed on Bitcoin *as is*. We also introduce a primitive that, at the price of minor Bitcoin extensions, supports *non-consensual* renegotiations, which are automatically triggered by the contract without requiring the participants' agreement.

Because of space constraints, we relegate part of the technicalities to [9].

2 BitML with renegotiation and recursion

We start by formalising contract preconditions. We use $\mathsf{A}, \mathsf{B}, \ldots$ to range over participants. We assume a set of *deposit names* $x, y, \ldots$, a set of deposit variables $d, e, \ldots$, and a set of *secret names* $a, b, \ldots$. We use $\chi, \chi', \ldots$ to range over deposit names and variables, and v, v' to range over non-negative values.

Definition 1 (Contract precondition). *Contract preconditions have the following syntax (the deposits χ in a contract precondition G must be distinct):*

$$
\begin{array}{rll}
G ::= & \mathsf{A} : v\,\mathtt{@}\,\chi & \textit{deposit of } v\text{฿} \textit{ put by } \mathsf{A} \\
\mid & \mathsf{A} : \mathtt{secret}\ a & \textit{secret committed by } \mathsf{A} \\
\mid & G \mid G & \textit{composition}
\end{array}
\qquad \diamond
$$

The precondition $\mathsf{A} : v\,\mathtt{@}\,\chi$ requires A to own v฿ in a deposit χ, and to spend it for stipulating the contract. The precondition $\mathsf{A} : \mathtt{secret}\ a$ requires A to generate

$C ::= \sum_{i \in I} D_i$	contract	$p ::= \mathit{true}$	truth
$D ::=$	guarded contract	$\mid p \wedge p$	conjunction
$\texttt{reveal}\ \boldsymbol{a}\ \texttt{if}\ p.\, C$	reveal secrets (if p is true)	$\mid \neg p$	negation
$\mid \texttt{withdraw}\ \mathsf{A}$	transfer the balance to A	$\mid E = E$	equality
$\mid \texttt{split}\ \boldsymbol{v} \to \boldsymbol{C}$	split the balance	$\mid E < E$	less than
$\mid \mathsf{A} : D$	wait for A's authorization	$E ::= \mathcal{E}$	static expression
$\mid \texttt{after}\ \mathcal{E} : D$	wait until time $\mathcal{E}$	$\mid a$	secret
$\mid * : \texttt{rngt}\ \mathtt{X}\langle \boldsymbol{\mathcal{E}} \rangle$	renegotiate the contract	$\mid E + E$	addition
		$\mid E - E$	subtraction

Fig. 1: Syntax of BitML contracts.

a secret a, and commit to it before the contract starts. After stipulation, A can choose whether to disclose the secret a, or not.

To define contracts, we assume a set of recursion variables, ranged over by $\mathtt{X}, \mathtt{Y}, \ldots$, and a language of *static expressions* $\mathcal{E}, \mathcal{E}', \ldots$, formed by integer constants k, integer variables $\alpha, \beta, \ldots$, and the usual arithmetic operators. We omit to define the syntax and semantics of static expressions, since they are standard. We assume that a closed static expression evaluates to a 32-bit value. We use the bold notation for sequences, e.g. $\boldsymbol{x}$ denotes a finite sequence of deposit names.

Definition 2 (Contract). *Contracts are terms with the syntax in Figure 1, where: (i) each recursion variable* $\mathtt{X}$ *has a unique defining equation* $\mathtt{X}(\boldsymbol{\alpha}) = \{G\}C$*; (ii) renegotiations* $* : \texttt{rngt}\ \mathtt{X}\langle \boldsymbol{\mathcal{E}} \rangle$ *have the correct number of arguments; (iii) the names* $\boldsymbol{a}$ *in* $\texttt{reveal}\ \boldsymbol{a}\ \texttt{if}\ p$ *are distinct, and they include those occurring in* p*; (iv) in a prefix* $\texttt{split}\ \boldsymbol{v} \to \boldsymbol{C}$*, the sequences* $\boldsymbol{v}$ *and* $\boldsymbol{C}$ *have the same length. We denote with* 0 *the empty sum. We assume that the order of decorations is immaterial, e.g.,* $\texttt{after}\ \mathcal{E} : \mathsf{A} : \mathsf{B} : D$ *is equivalent to* $\mathsf{B} : \mathsf{A} : \texttt{after}\ \mathcal{E} : D$*.* ◇

A contract C is a choice among guarded contracts D_i. A guarded contract $\texttt{reveal}\ \boldsymbol{a}\ \texttt{if}\ p.\, C'$ continues as C' once all the secrets $\boldsymbol{a}$ have been revealed and satisfy the predicate p. The guarded contract $\texttt{split}\ (v_1 \to C_1 \mid \cdots \mid v_n \to C_n)$ divides the contract into n contracts C_i, each one with balance v_i. The sum of the v_i must coincide with the current balance. The action $\texttt{withdraw}\ \mathsf{A}$ transfers the whole balance to A. When enabled, the above actions can be fired by anyone at anytime. To restrict *who* can execute a branch and *when*, one can use the decoration $\mathsf{A} : D$, requiring to wait for A's authorization, and the decoration $\texttt{after}\ \mathcal{E} : D$, requiring to wait until the time specified by the static expression $\mathcal{E}$. The action $* : \texttt{rngt}\ \mathtt{X}\langle \boldsymbol{\mathcal{E}} \rangle$ allows the participants involved in the contract to renegotiate it. Intuitively, if $\mathtt{X}(\boldsymbol{\alpha}) = \{G\}C$, then the contract continues as $C\{\boldsymbol{\mathcal{E}}/\boldsymbol{\alpha}\}$ if all the participants give their authorization, and satisfy the precondition G.

Definition 3 (Contract advertisement). *A contract advertisement is a term* $\{G\}C$ *such that: (i) each secret name in* C *occurs in* G*; (ii)* G *requires a deposit*

from each A *in* $\{G\}C$*; (iii) each* $* : \mathtt{rngt}\ \mathtt{X}\langle\boldsymbol{\mathcal{E}}\rangle$ *in* C *refers to a defining equation* $\mathtt{X}(\boldsymbol{\alpha}) = \{G'\}C'$ *where the participants in* G' *are the same as those in* G*.* $\diamond$

The second condition is used to guarantee that the contract is stipulated only if *all* the involved participants give their authorizations. The last condition is only used to simplify the technical development. We outline in Section 5 how to relax it, by allowing renegotiations to exclude some participants, or to include new ones, which were not among those who originally stipulated the contract.

We now extend the reduction semantics of BitML [11], by focussing on the new renegotiation primitive. Because of space limitations, here we just provide the underlying intuition, relegating the full formalisation to [9]. We start by defining the configurations of the semantics.

Definition 4 (Configuration). *Configurations have the following syntax:*

$\Gamma ::=$	0	*empty*
	$\mid \{G\}^x C$	*contract advertisement (name* x *is optional)*
	$\mid \langle C, v\rangle_x$	*active contract containing* v฿
	$\mid \langle \mathsf{A}, v\rangle_x$	*deposit of* v฿ *redeemable by* A
	$\mid \mathsf{A}[\chi]$	*authorization of* A *to perform action* χ
	$\mid \{\mathsf{A} : a\#N\}$	*committed secret of* A *(*$N \in \mathbb{N} \cup \{\bot\}$*)*
	$\mid \mathsf{A} : a\#N$	*revealed secret of* A *(*$N \in \mathbb{N}$*)*
	$\mid \mathsf{A} : d \leftarrow x$	A*'s deposit variable* d *assigned to deposit name* x
	$\mid \Gamma \mid \Gamma'$	*parallel composition*

We denote with $\Gamma \mid t$ *a* timed *configuration, where* $t \in \mathbb{N}$ *is a global time.* $\diamond$

We illustrate configurations and their semantics through a series of examples.

Deposits. A deposit $\langle \mathsf{A}, v\rangle_x$ can be subject to several operations, like e.g. split into two smaller deposits, join with another deposit, transfer to another participant, or destroy. In all cases A must authorise the operation. For instance, to authorize the join of two deposits, A can perform the following step:

$$\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \ \rightarrow\ \langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v'\rangle]$$

where $x, y \rhd \langle \mathsf{A}, v + v'\rangle$ is the authorization of A to spend x. After A also provides the dual authorization to spend y, anyone can actually join the deposits:

$$\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v'\rangle] \mid \mathsf{A}[y, x \rhd \langle \mathsf{A}, v + v'\rangle] \ \rightarrow\ \langle \mathsf{A}, v + v'\rangle_z$$

Advertisement. Any participant can broadcast a new contract advertisement $\{G\}C$, provided that all the deposits mentioned in G exist in the current configuration, and that the names of the secrets declared in G are fresh.

Stipulation. To stipulate an advertised contract $\{G\}C$, all the participants mentioned in it must fulfill the preconditions, and authorise the stipulation. For instance, let $G = \mathsf{A}{:}\,1\,@\,x \mid \mathsf{B}{:}\,1\,@\,y \mid \mathsf{A} : \mathtt{secret}\,a$, and let C be an arbitrary contract involving only A and B. The stipulation starts from a configuration containing the advertisement and the participants' deposits:

$$\Gamma \;=\; \{G\}C \mid \langle \mathsf{A}, 1\rangle_x \mid \langle \mathsf{B}, 1\rangle_y$$

At this point the participants must commit to their secrets (in this case, only A has a secret). This is rendered as a sequence of steps:

$$\Gamma \;\to^*\; \Gamma \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \rhd \{G\}C] \mid \mathsf{B}[\# \rhd \{G\}C] \;=\; \Gamma'$$

where $\{\mathsf{A} : a\#N\}$ represents the fact that A has committed to the secret N, while $\mathsf{A}[\# \rhd \{G\}C]$ and $\mathsf{B}[\# \rhd \{G\}C]$ represent ending the commitment phase (these steps might seem redundant, but they are useful to obtain a step-by-step correspondence between BitML executions and Bitcoin executions).

After that, A and B must perform an additional sequence of steps to authorize the transfer of their deposits x, y to the contract:

$$\Gamma' \;\to^*\; \Gamma' \mid \mathsf{A}[x \rhd \{G\}C] \mid \mathsf{B}[y \rhd \{G\}C] \;=\; \Gamma''$$

where $\mathsf{A}[x \rhd \{G\}C]$ and $\mathsf{B}[y \rhd \{G\}C]$ are the authorizations to spend x and y.

At this point all the needed authorizations have been given, so the advertisement can be turned into an active contract. This step consumes the deposits and all the authorizations, and creates an active contract, with a fresh name z:

$$\Gamma'' \;\to\; \langle C, 2\rangle_z \mid \{\mathsf{A} : a\#N\}$$

Renegotiation. We illustrate the steps to renegotiate $\mathtt{X}\langle\alpha\rangle = \{G\}C$, where $G = \mathsf{A}{:}\,1\,@\,d \mid \mathsf{B}{:}\,1\,@\,e \mid \mathsf{A} : \mathtt{secret}\,a$, and C is an arbitrary contract involving only A and B, and possibly containing the integer variable α in static expressions. Here, G requires A and B to spend two 1B̈ deposits, and A to commit to a secret. Unlike in the case of contract stipulation above, deposits names are unknown before renegotiation, so we use the deposit variables d, e to refer to them.

Consider a configuration $\langle * : \mathtt{rngt}\ \mathtt{X}\langle k\rangle + C'', v\rangle_x \mid \Gamma$, where C'' contains the branches alternative to the renegotiation. A possible execution of the action $* : \mathtt{rngt}\ \mathtt{X}\langle k\rangle$ starts as follows:

$$\langle * : \mathtt{rngt}\ \mathtt{X}\langle k\rangle + C'', v\rangle_x \mid \Gamma \;\to\; \langle * : \mathtt{rngt}\ \mathtt{X}\langle k\rangle + C'', v\rangle_x \mid \{G'\}^x C' \mid \Gamma \;=\; \Gamma'$$

where the advertisement $\{G'\}^x C'$ is obtained by transforming $\{G\}C$ as follows: (i) variables d, e are renamed into fresh ones d', e', and similarly the secret name a into a', (ii) the static expressions in C are evaluated, assuming $\alpha = k$, and replaced with their results. The superscript x in the advertisement is used to record that, when the renegotiation is concluded, the contract x must be reduced.

In the subsequent steps participants choose the actual deposit names, and A commits to her secret. If A owns in Γ a deposit $\langle \mathsf{A}, 1\rangle_y$, she can choose $d' = y$

to satisfy the precondition G. Similarly, B can choose $e' = z$ if he owns such a deposit in Γ. These choices are performed as follows:

$$\begin{aligned}\Gamma' \;\to^*\; \Gamma' \mid \mathsf{A} : d' \leftarrow y \mid \{\mathsf{A} : a' \# N\} \mid \mathsf{A}[\# \rhd \{G'\}^x C'] \\ \mid \mathsf{B} : e' \leftarrow z \mid \mathsf{B}[\# \rhd \{G'\}^x C'] \qquad\qquad = \; \Gamma''\end{aligned}$$

At this point, participants must authorise spending their deposits and the balance of the contract at x. This is done through a series of steps:

$$\begin{aligned}\Gamma'' \;\to^*\; \Gamma'' \mid \mathsf{A}[y \rhd \{G'\}^x C'] \mid \mathsf{A}[x \rhd \{G'\}^x C'] \\ \mid \mathsf{B}[z \rhd \{G'\}^x C'] \mid \mathsf{B}[x \rhd \{G'\}^x C'] \;=\; \Gamma'''\end{aligned}$$

Finally, the new contract is stipulated. This closes the old contract, and transfers its balance to the newly generated one, with a fresh name x':

$$\Gamma''' \;\to\; \langle C', v + 2\rangle_{x'} \mid \Gamma$$

Note that the branches in C'' are discarded only in the last step above, where we complete the renegotiation. Before this step, it would have been possible to take one of the branches in C'', aborting the renegotiation.

Withdraw. Executing `withdraw` A transfers the whole contract balance to A:

$$\langle \texttt{withdraw}\ \mathsf{A} + C', v\rangle_x \;\to\; \langle \mathsf{A}, v\rangle_y$$

After the execution, the alternative branch C' is discarded, and a fresh deposit of vɃ for A is created. Note that the active contract x is terminated.

Split. The `split` primitive divides the contract balance in n parts, each one controlled by its own contract. For instance, if $n = 2$:

$$\langle (\texttt{split}\ v_1 \to C_1 \mid v_2 \to C_2) + C', v_1 + v_2\rangle_x \;\to\; \langle C_1, v_1\rangle_y \mid \langle C_2, v_2\rangle_z$$

After this step, the new spawned contracts C_1 and C_2 are executed concurrently.

Reveal. The prefix `reveal` $\boldsymbol{a}$ `if` p can be fired if all the committed secrets $\boldsymbol{a}$ have been revealed, and satisfy the guard p. For instance, if $\Gamma = \mathsf{A} : a \# N \mid \mathsf{B} : b \# N$:

$$\langle (\texttt{reveal}\ ab\ \texttt{if}\ a = b.\, C) + C', v\rangle_x \mid \Gamma \;\to\; \langle C, v\rangle_y \mid \Gamma$$

The terms $\mathsf{A} : a \# N$ and $\mathsf{B} : b \# N$ represent the fact that the secrets a and b have been revealed. Crucially, only the participant who performed the commitment can add the corresponding term to the configuration.

Authorizations. A branch decorated by $\mathsf{A} : \cdots$ can be taken only if the participant A has provided her authorization. For instance:

$$\langle \mathsf{A} : \texttt{withdraw}\ \mathsf{B} + C', v\rangle_x \mid \mathsf{A}[x \rhd \mathsf{A} : \texttt{withdraw}\ \mathsf{B}] \;\to\; \langle \mathsf{B}, v\rangle_y$$

The leftmost configuration contains the term $\mathsf{A}[x \rhd \mathsf{A} : \texttt{withdraw}\ \mathsf{B}]$, which represents A's authorization to take the branch `withdraw` B. This enables the step to be taken. When multiple authorizations are required, the branch can be taken only after all of them occur in the configuration.

Time constraints. We represent time in configurations as $\Gamma \mid t$, where Γ is the untimed part of the configuration and t is the current time. We always allow the time to advance through the rule $\Gamma \mid t \to \Gamma \mid t + \delta$, for all $\delta > 0$. A branch decorated with $\texttt{after}\, d$ can be taken only if time d has passed. For instance:

$$\langle \texttt{after}\, d : \texttt{withdraw}\ \mathsf{B}, v \rangle_x \mid t \ \to\ \langle \mathsf{B}, v \rangle_y \mid t \qquad \text{if } t \geq d$$

For the branches not guarded by `after`, we lift transitions from untimed to timed configurations: namely, for an untimed transition $\Gamma \to \Gamma'$, we also have the timed transition $\Gamma \mid t \to \Gamma' \mid t$. This reflects the assumption that participants can always meet deadlines, if they want to.

3 Executing BitML on Bitcoin

To execute a BitML contract, participants first compile it to a set of Bitcoin transactions, and then append these transactions to the blockchain, each following their own strategy. Participants' strategies can involve other actions besides appending transactions, like e.g. broadcasting signatures on given transactions (which corresponds, in BitML, to add an authorization to the configuration), revealing secrets, and waiting some time (see Definition 16 in [11]). The coherence between the BitML semantics and the execution on Bitcoin is guaranteed by a step-by-step correspondence between the transitions of the BitML semantics and the actions performed by participants on the Bitcoin network.

In this section we illustrate the compiler and the execution protocol through a couple of examples, focussing on the new renegotiation primitive. The needed background on Bitcoin will be introduced along with these examples. We relegate the formal definition of the compilation rules to [9].

Zero-coupon bond. Recall the *ZCB* contract from Section 1:

$$\mathit{ZCB} \ =\ \texttt{split}\ \big(1 \to \texttt{withdraw}\ \mathsf{B} \ \mid\ 2 \to \texttt{after}\, 2030 : \texttt{withdraw}\ \mathsf{A}\big)$$

The precondition $\mathsf{A} : 1\,\texttt{@}\,x_1 \mid \mathsf{B} : 2\,\texttt{@}\,x_2$ requires A to deposit 1฿ in the contract, and B to deposit 2฿. In Bitcoin, this precondition corresponds to requiring two unspent transactions redeemable by A and B, and containing the required amounts. We represent these transactions as follows, using the notation in [7]:

T_{x_1}
in: $\cdots$
wit: $\cdots$
out: $(\lambda x.\mathsf{versig}_{\mathbf{K}(\mathsf{A})}(x), 1\text{฿})$

T_{x_2}
in: $\cdots$
wit: $\cdots$
out: $(\lambda x.\mathsf{versig}_{\mathbf{K}(\mathsf{B})}(x), 2\text{฿})$

The transaction T_{x_1} is a record with three fields (T_{x_2} is similar). The in field points to one or more previous transactions in the blockchain. The field out is a pair, whose first element is a boolean predicate (with parameter x), and the second element, 1฿, is the amount that a subsequent transaction satisfying the predicate can redeem from T_{x_1}. Here, the predicate $\mathsf{versig}_{\mathbf{K}(\mathsf{A})}(x)$ is true when x is a signature of A on the redeeming transaction (i.e., one having T_{x_1} as in).

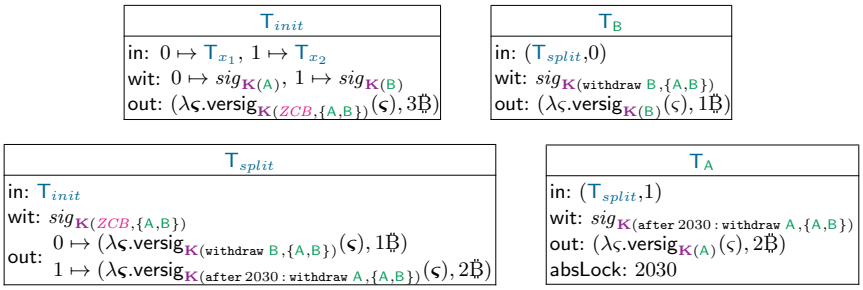

Fig. 2: Transactions obtained by compiling the *ZCB* contract.

The contract *ZCB* is compiled into the transactions in Figure 2. The first one that can be appended to the blockchain is T_{init}. This requires a few conditions to be met: (i) T_{x_1} and T_{x_2} are *unspent* on the blockchain, i.e. no other transactions spend them; (ii) the amount specified in the out field of T_{init} does not exceed the sum of the amounts in T_{x_1} and T_{x_2}; (iii) the predicates in the out fields of T_{x_1} and T_{x_2} are true, after replacing the formal parameters with the signatures $sig_{\mathbf{K}(\mathsf{A})}$ and $sig_{\mathbf{K}(\mathsf{B})}$, contained in the wit field of T_{init}. The contract *ZCB* becomes stipulated once T_{init} is on the blockchain.

After that, the `split` action can be performed by either A or B, by redeeming T_{init} with T_{split}. This transaction uses $\mathbf{K}(ZCB, \{\mathsf{A}, \mathsf{B}\})$, a set of two key pairs, each one owned by each participant. These keys are only used in this step, to ensure that no transaction but T_{split} can redeem T_{init}.

The transaction T_{split} creates two unspent outputs (indexed by 0 and 1), corresponding to the two parallel components of the `split`, each with its own balance. These outputs can be redeemed independently, by different transactions. The output at index 0 can only be redeemed by T_{B} (note that T_{B}'s in field refers to the output 0 of T_{split}), transferring $1\ddot{\mathrm{B}}$ to B. No other redemption is possible, since such output requires a signature with a specific key set, i.e. $\mathbf{K}(\mathtt{withdraw}\ \mathsf{B}, \{\mathsf{A}, \mathsf{B}\})$, which is not used for any other purpose. Further, the output of T_{B} can be redeemed with B's key, without A's one. Similarly, the output 1 of T_{split} can be redeemed by T_{A}, which in turns transfers $2\ddot{\mathrm{B}}$ to A. The absLock field in T_{A} ensures that this may only happen after time 2030.

The stipulation protocol followed by participants requires that all the signatures needed to append the transactions in Figure 2 are exchanged *before* T_{init} is appended. This is obtained by exchanging the signatures of T_{init} after all the other signatures. This ensures that, once the execution of *ZCB* starts, any honest participant can make it proceed, by appending a transaction that correspond to any of the enabled BitML actions.

In general, to guarantee that such liveness property holds, the contract must be suitably crafted, using the $D + \mathtt{after}\, t : D'$ pattern discussed in Section 1. In Section 6 we discuss techniques to statically verify this property.

Zero-coupon bond with renegotiation. Compiling *ZCB2* yields the transactions:

T_{init}	
in:	T_{x_1}
wit:	$sig_{\mathbf{K}(\mathsf{A})}$
out:	$(\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(ZCB2,\{\mathsf{A},\mathsf{B}\})}(\varsigma),$ $1\ddot{\mathrm{B}})$

T_{split}	
in:	T_{init}
wit:	$sig_{\mathbf{K}(ZCB2,\{\mathsf{A},\mathsf{B}\})}$
out:	$0 \mapsto (\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(\mathtt{withdraw}\ \mathsf{B},\{\mathsf{A},\mathsf{B}\})}(\varsigma), 1\ddot{\mathrm{B}})$
	$1 \mapsto (\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(*:\mathtt{rngt}\ \mathtt{X}\langle\rangle,\{\mathsf{A},\mathsf{B}\})}(\varsigma), 0\ddot{\mathrm{B}})$

T_{B}	
in:	$(\mathsf{T}_{split},0)$
wit:	$sig_{\mathbf{K}(\mathtt{withdraw}\ \mathsf{B},\{\mathsf{A},\mathsf{B}\})}$
out:	$(\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(\mathsf{B})}(\varsigma),$ $1\ddot{\mathrm{B}})$

Once these three transactions are on the blockchain, the only enabled action in the corresponding BitML contract is $* : \mathtt{rngt}\ \mathtt{X}\langle\rangle$, which asks $2\ddot{\mathrm{B}}$ from B as a precondition. At the Bitcoin level, satisfying this precondition requires B to broadcast the identifier of a transaction T_y holding $2\ddot{\mathrm{B}}$ and redeemable by himself. In BitML, this corresponds to choosing the deposit name y for the deposit variable d. Then, participants compile the contract advertisement $\{\mathsf{B}: 2\,@\,d\}C$, where $C = \mathtt{after}\,2030 : \mathtt{withdraw}\ \mathsf{A}$, after replacing d with y. The compiler produces the following transactions:

$\mathsf{T}^{\mathtt{X}\langle\rangle}_{init}$	
in:	$0 \mapsto (\mathsf{T}_{split}, 1), 1 \mapsto \mathsf{T}_y$
wit:	$0 \mapsto sig_{\mathbf{K}(*:\mathtt{rngt}\ \mathtt{X}\langle\rangle,\{\mathsf{A},\mathsf{B}\})}$
	$1 \mapsto sig_{\mathbf{K}(\mathsf{B})}$
out:	$(\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(C,\{\mathsf{A},\mathsf{B}\})}(\varsigma), 2\ddot{\mathrm{B}})$

T_{A}	
in:	$\mathsf{T}^{\mathtt{X}\langle\rangle}_{init}$
wit:	$sig_{\mathbf{K}(C,\{\mathsf{A},\mathsf{B}\})}$
out:	$(\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(\mathsf{A})}(\varsigma), 2\ddot{\mathrm{B}})$
absLock:	2030

The renegotiation succeeds once $\mathsf{T}^{\mathtt{X}\langle\rangle}_{init}$ is on the blockchain. After that, any participant can perform the $\mathtt{withdraw}\ \mathsf{A}$, by appending T_{A} to the blockchain.

As shown by this example, the compiler handles renegotiation as follows:

- at stipulation time, it does not produce transactions for any $* : \mathtt{rngt}\ \mathtt{X}\langle\rangle$;
- at renegotiation time, the participants broadcast the identifiers of their new deposits, and the commitments of their new secrets. Static expressions are then evaluated, and replaced by their value. Finally, the new contract is compiled as usual, with the exception that the new initial transaction has an extra input, which transfers the balance of the caller contract to the callee (in the *ZCB2* example, this extra input is $(\mathsf{T}_{split}, 1)$ within $\mathsf{T}^{\mathtt{X}\langle\rangle}_{init}$).

Computational soundness The main result of [11] is computational soundness, which ensures that each execution trace at the Bitcoin level has a corresponding one in the semantics of BitML. This was achieved by formalizing the semantics of Bitcoin using a computational model, where participants can exchange bitstrings as messages, and append transactions to the blockchain. Then, a coherence relation was defined to relate symbolic runs to computational ones, essentially matching symbolic moves with their implementation in Bitcoin.

Our extension of BitML with renegotiation still enjoys computational soundness. The argument is similar, and requires extending the coherence relation to the new primitive. In particular, the reduction:

$$\{G\}^x C \mid \Gamma \;\to\; \{G\}^x C \mid \Gamma \mid \;\|_i \{\mathsf{A} : a_i \# N_i\} \mid \;\|_j \mathsf{A} : d_j \leftarrow x_j \mid \mathsf{A}[\# \rhd \{G\}^x C]$$

corresponds, in Bitcoin, to A broadcasting a message which contains the hashes of her secrets and the transaction identifiers that she wishes to use as deposits.

Instead, the reduction:

$$\{G\}^x C \mid \Gamma \;\to\; \{G\}^x C \mid \Gamma \mid \mathsf{A}[x \rhd \{G\}^x C]$$

corresponds to A signing all the transactions obtained by compiling the new contract, and broadcasting the signatures. A participant signs T_{init} only after receiving the signatures of the other transactions from all the other participants.

Computational soundness requires that each contract involves at least one participant, say A, who follows the Bitcoin implementation of BitML. In particular, A follows the stipulation and renegotiation protocols correctly, i.e. signing nothing but the protocol messages, and signing T_{init} last. We also make the usual assumptions on computational adversaries: they can only run PPTIME algorithms, and they can break the underlying cryptography with negligible probability, only. Consequently, we only consider computational runs of polynomial length (with respect to the security parameter). This is because in longer runs the adversary would be able to break the cryptography by brute force.

Below, we provide an intuitive statement of computational soundness. The formal statement is in [9].

Theorem 1 (Computational soundness). *Under the hypotheses above, each Bitcoin-level computational run has a corresponding coherent BitML run, with overwhelming probability.*

4 A fair recursive coin flipping game

To illustrate recursion in our extended BitML, we introduce a simple game where two players repeatedly flip coins, and the one who wins two consecutive flips takes the pot. The precondition requires each player to deposit 3฿ and choose a secret:

$$\mathsf{A}:3\,\texttt{@}\,x \;\mid\; \mathsf{A}:\texttt{secret}\,a \;\mid\; \mathsf{B}:3\,\texttt{@}\,y \;\mid\; \mathsf{B}:\texttt{secret}\,b$$

The contract *CFG* (Figure 3) asks B to reveal his secret first: if B waits too much, A can withdraw the contract funds after time 1. Then, it is A's turn to reveal (before time 2, otherwise B can withdraw the funds). The current flip winner is A if the secrets of A and B are equal, otherwise it is B. At this point, the contract can be renegotiated as $\mathtt{X}_{\mathsf{A}}\langle 1\rangle$ or $\mathtt{X}_{\mathsf{B}}\langle 1\rangle$, depending on the flip winner (the parameter 1 represents the round). If players do not agree on the renegotiation, then the funds are split fairly, according to the current expected win.

The contract $\mathtt{X}_{\mathsf{A}}\langle n\rangle$ requires A and B to generate fresh secrets for the n-th turn. If A wins again, she can withdraw the pot, otherwise the contract can be renegotiated as $\mathtt{X}_{\mathsf{B}}\langle n+1\rangle$. If the players do not agree on the renegotiation, the pot is split fairly between them. The contract $\mathtt{X}_{\mathsf{B}}$ is similar.

The following theorem states that our coin flipping game is fair. Fairness ensures that the expected payoff of a *rational* player is always non-negative, notwithstanding the behaviour of the other player. Rational players must choose random secrets in $\{0,1\}$. Indeed, non uniformly distributed secrets can make

$$
\begin{aligned}
\mathit{CFG} = \;& \texttt{reveal}\, b\, \texttt{if}\, 0 \le b \le 1.\big(\\
& \quad \texttt{reveal}\, ab\, \texttt{if}\, a = b.\, (* : \texttt{rngt}\, \mathtt{X}_{\mathsf{A}}\langle 1\rangle \;+\; \texttt{after}\, 3 : \mathit{Split}_{\mathsf{A}}) \\
& \quad + \texttt{reveal}\, ab\, \texttt{if}\, a \neq b.\, (* : \texttt{rngt}\, \mathtt{X}_{\mathsf{B}}\langle 1\rangle \;+\; \texttt{after}\, 3 : \mathit{Split}_{\mathsf{B}}) \\
& \quad + \texttt{after}\, 2 : \texttt{withdraw}\, \mathsf{B}\big) \\
& + \texttt{after}\, 1 : \texttt{withdraw}\, \mathsf{A} \\
\mathtt{X}_{\mathsf{A}}\langle n\rangle = \;& \{\mathsf{A} : \texttt{secret}\, a \mid \mathsf{B} : \texttt{secret}\, b\} \\
& \quad \texttt{reveal}\, b\, \texttt{if}\, 0 \le b \le 1.\big(\\
& \quad \quad \texttt{reveal}\, ab\, \texttt{if}\, a = b.\, \texttt{withdraw}\, \mathsf{A} \\
& \quad + \texttt{reveal}\, ab\, \texttt{if}\, a \neq b.\, (* : \texttt{rngt}\, \mathtt{X}_{\mathsf{B}}\langle n+1\rangle \;+\; \texttt{after}\, (3n+3) : \mathit{Split}_{\mathsf{B}}) \\
& \quad + \texttt{after}\, (3n+2) : \texttt{withdraw}\, \mathsf{B}\big) \\
& + \texttt{after}\, (3n+1) : \texttt{withdraw}\, \mathsf{A} \\
\mathtt{X}_{\mathsf{B}}\langle n\rangle = \;& \{\mathsf{A} : \texttt{secret}\, a \mid \mathsf{B} : \texttt{secret}\, b\} \\
& \quad \texttt{reveal}\, b\, \texttt{if}\, 0 \le b \le 1.\big(\\
& \quad \quad \texttt{reveal}\, ab\, \texttt{if}\, a = b.\, (* : \texttt{rngt}\, \mathtt{X}_{\mathsf{A}}\langle n+1\rangle \;+\; \texttt{after}\, (3n+3) : \mathit{Split}_{\mathsf{A}}) \\
& \quad + \texttt{reveal}\, ab\, \texttt{if}\, a \neq b.\, \texttt{withdraw}\, \mathsf{B} \\
& \quad + \texttt{after}\, (3n+2) : \texttt{withdraw}\, \mathsf{B}\big) \\
& + \texttt{after}\, (3n+1) : \texttt{withdraw}\, \mathsf{A} \\
\mathit{Split}_{\mathsf{A}} = \;& \texttt{split}\, (4 \to \texttt{withdraw}\, \mathsf{A} \mid 2 \to \texttt{withdraw}\, \mathsf{B}) \\
\mathit{Split}_{\mathsf{B}} = \;& \texttt{split}\, (4 \to \texttt{withdraw}\, \mathsf{B} \mid 2 \to \texttt{withdraw}\, \mathsf{A})
\end{aligned}
$$

Fig. 3: A recursive coin flipping game.

the adversary bias the coin flip in her favour. Further, choosing a secret different from 0 or 1 would decrease the player payoff. Indeed, B would be prevented from revealing his secrets (by the predicate in the `reveal` b), and so A could win after the timeout. If A chooses a secret different from 0 or 1, she makes B win the round (since B wins when the secrets are different). Rationality also requires to reveal secrets in time (before the alternative `after` branch is enabled), and to take the *Split* branch if restipulation does not occur in time. This ensures that, when renegotiation happens, there is still time to reveal the round secrets. Indeed, a late renegotiation could enable the other player to win by timeout.

Theorem 2. *The expected payoff of a rational player is always non-negative.*

Proof (Sketch). First, we consider the case where renegotiation always happens. A rational player wins each coin flip with probability $1/2$, at least: so, the probability of winning the whole game is also $1/2$, at least. In the general case, the renegotiation at the end of each round may fail. When this happens, the rational player takes the *Split* branch, distributing the pot according to the expected payoff in the *current* game state, thus ensuring the fairness of the whole game. The player who won the last coin flip is expected to win p₿, with $p = 1/2 \cdot 6 + 1/2 \cdot (1/2 \cdot p + 1/2 \cdot 0)$, giving $p = 4$. Accordingly, the *Split* contracts transfer 4₿ to the winner of the last flip and $(6-4)$₿ $= 2$₿ to the other player.

5 More expressive renegotiation primitives

The renegotiation primitive we have proposed for BitML is motivated by its simplicity, and by the possibility of compiling into standard Bitcoin transactions. By adding some degree of complexity, we can devise more general primitives, which could be useful in certain scenarios. We discuss below some alternatives.

Renegotiation-time parameters. The primitive $* : \texttt{rngt X}\langle \boldsymbol{\mathcal{E}} \rangle$ allows participants to choose at run-time only the deposit variables used in the renegotiated contracts, and to commit to new secrets. A possible extension is to allow participants to choose at run-time *arbitrary* values for the renegotiation parameters $\boldsymbol{\mathcal{E}}$.

For instance, consider a mortgage payment, where a buyer A must pay 10฿ to a bank B in 10 installments. After A has paid the first five installments (of 1฿ each), the bank might propose to renegotiate the contract, varying the amount of the installment. Using the BitML renegotiation primitive presented in Section 2, we could not model this contract, since the new amount and the number of installments are unknown at the time of the original stipulation. Technically, the issue is that the primitive $* : \texttt{rngt X}\langle \boldsymbol{\mathcal{E}} \rangle$ only involves static expressions $\mathcal{E}$, the value of which is determined at stipulation time.

To cope with non-statically known values, we could extend guarded contracts with terms of the form $* : \texttt{rngt X}\langle \mathsf{B} : v \rangle$, declaring that the value v is to be chosen by B at renegotiation time. For instance, this would allow to model our installments payment plan as $\texttt{IPP}\langle 1 \rangle$, with the following defining equations:

$$\begin{aligned}
\texttt{IPP}\langle \alpha < 5 \rangle &= \{\mathsf{A}: 1 \texttt{@} d\}\big(\texttt{split } 1 \to \texttt{withdraw B} \mid 0 \to * : \texttt{rngt IPP}\langle \alpha + 1 \rangle\big) \\
\texttt{IPP}\langle 5 \rangle &= \{\mathsf{A}: 1 \texttt{@} d\}\big(\texttt{split } 1 \to \texttt{withdraw B} \mid 0 \to * : \texttt{rngt Y}\langle \mathsf{B} : k, \mathsf{B} : v \rangle\big) \\
\texttt{Y}\langle \alpha \neq 1, \beta \rangle &= \{\mathsf{A}: \beta \texttt{@} d\}\big(\texttt{split } \beta \to \texttt{withdraw B} \mid 0 \to * : \texttt{rngt Y}\langle \alpha - 1, \beta \rangle\big) \\
\texttt{Y}\langle 1, \beta \rangle &= \{\mathsf{A}: \beta \texttt{@} d\}\ \texttt{withdraw B}
\end{aligned}$$

where in $\texttt{IPP}\langle 5 \rangle$, the bank chooses the number of installments k, as well as the amount v of each installment. Note that if A does not agree with these values, the renegotiation fails. A more refined version of the contract should take this possibility into account, by adding suitable compensation branches. Although adding the new primitive would moderately increase the complexity of the semantics and of the compiler, this extension can still be implemented on top of standard Bitcoin, preserving our computational soundness result.

Renegotiation with a given set of participants. As we have remarked in Section 2, a renegotiation can be performed only if *all* the participants of the contract agree. To generalise, we could require the agreement of a *given* set of participants (possibly, not among those who originally stipulated the contract).

For instance, consider an escrow service between a buyer A and a seller B for the purchase of an item worth 1฿. The normal case is that the buyer authorizes the transfer of 1฿ after receiving the item, but it may happen that a dishonest seller never sends the item, or that a dishonest buyer never authorizes

the payment. To cope with these cases, the participants can renegotiate the contract, including an escrow service M which mediates the dispute, as follows:

$$\mathsf{A} : \texttt{withdraw}\ \mathsf{B}\ +\ \mathsf{B} : \texttt{withdraw}\ \mathsf{A} +\ \mathsf{A} : \mathsf{M} : \texttt{rngt}\ \texttt{Refd}_{\mathsf{A}}\langle\rangle\ +\ \mathsf{B} : \mathsf{M} : \texttt{rngt}\ \texttt{Refd}_{\mathsf{B}}\langle\rangle$$

$$\texttt{Refd}_{\mathsf{P}} = \{\mathsf{P}{:}\,0.1\,@\,d\}\ \texttt{split}\ \big(0.1 \to \texttt{withdraw}\ \mathsf{M} \mid 1 \to \texttt{withdraw}\ \mathsf{P}\big)$$

where $\mathsf{A} : \mathsf{M} : \texttt{rngt}\ \texttt{Refd}_{\mathsf{A}}\langle\rangle$ means that only A and M need to agree in order for the contract $\texttt{Refd}_{\mathsf{A}}$ to be executed, resolving the dispute. In this case it is crucial that the renegotiation is possible even without the agreement between A and B. Indeed, if M decides to refund A (by authorizing $\texttt{Refd}_{\mathsf{A}}$), it is not to be expected that also B agrees. Similarly to the one discussed before, also this extension can be implemented on-top of Bitcoin. The computational soundness property is preserved, under the assumption that at least one participant in any renegotiation is *honest*, i.e. it follows the renegotiation protocol. Crucially, if a renegotiation only involves dishonest participants, the renegotiated contract could be anything, not necessarily that prescribed in the original contract.

Non-consensual renegotiation. In the variants of $*$: rngt discussed before, renegotiation requires one or more participants to agree. Hence, each use of $*$: rngt must include suitable alternative branches, to be fired in case the renegotiation fails. In certain scenarios, we may want to renegotiate the contract without the participants having to agree. To this purpose, we can introduce a new primitive $\texttt{call}\ \texttt{X}\langle\rangle$, which continues as $\texttt{X}\langle\rangle$ without requiring anyone to agree. For simplicity, we assume the defining equations of this primitive of the form $\texttt{X}(\boldsymbol{\alpha}) = \{v\}C$, where v represents the amount of Ƀ added to the contract, by anyone.

We exemplify the new primitive to design a two-players game which starts with a bet of 1Ƀ from A, and a bet of 2Ƀ from B. Then, starting from A, players take turns adding 2Ƀ each to the pot. The first one who is not able to provide the additional 2Ƀ within a given time loses the game, allowing the other player to take the whole pot. The contract is as follows:

$$\begin{aligned} C &= \{\mathsf{A}{:}\,1\,@\,x \mid \mathsf{B}{:}\,2\,@\,y\}(\texttt{call}\ \texttt{X}_{\mathsf{A}}\langle 2\rangle + \texttt{after}\ 1 : \texttt{withdraw}\ \mathsf{B}) \\ \texttt{X}_{\mathsf{A}}\langle n\rangle &= \{2\}(\texttt{call}\ \texttt{X}_{\mathsf{B}}\langle n+1\rangle + \texttt{after}\ n : \texttt{withdraw}\ \mathsf{A}) \\ \texttt{X}_{\mathsf{B}}\langle n\rangle &= \{2\}(\texttt{call}\ \texttt{X}_{\mathsf{A}}\langle n+1\rangle + \texttt{after}\ n : \texttt{withdraw}\ \mathsf{B}) \end{aligned}$$

Unlike $*$: rngt, the action call can be fired without the authorizations of all the players: it just requires that the authorization to gather 2Ƀ is provided, by anyone. Even though the sender of these 2Ƀ is not specified in the contract, it is implicit in the game mechanism: for instance, when $\texttt{X}_{\mathsf{A}}\langle n\rangle$ calls $\texttt{X}_{\mathsf{B}}\langle n+1\rangle$, only B is incentivized to add 2Ƀ, since not doing so will make A win.

Implementing the call primitive on top of Bitcoin seems unfeasible: even if it were possible to use complex off-chain multiparty computation protocols [16], doing so might be impractical. Rather, we would like to extend Bitcoin as much as needed for the new primitive. In our implementation of BitML, we compile contracts to sets of transactions and make participants sign them. In

standard BitML this is doable since, at stipulation time, we can finitely over-approximate the reducts of the original contract. Recursion can make this set infinite, e.g. $\mathtt{X}_{\mathsf{A}}\langle 2\rangle, \mathtt{X}_{\mathsf{A}}\langle 3\rangle, \ldots$, hence impossible to compile and sign statically. A way to cope with this is to extend Bitcoin with *malleable* signatures which only cover the part of the transaction not affected by the parameter n in $\mathtt{X}_{\mathsf{B}}\langle n\rangle$. Further, signatures must not cover the in fields of transactions, since they change as recursion unfolds. In this way, the same signature can be reused for each call.

Adding malleability provides flexibility, but poses some risks. For instance, instead of redeeming the transaction corresponding to $\mathtt{X}_{\mathsf{A}}\langle n\rangle$ with the transaction of $\mathtt{X}_{\mathsf{B}}\langle n+1\rangle$ one could instead use the transaction of $\mathtt{X}_{\mathsf{B}}\langle n+100\rangle$, since the two transactions have the same signature. To overcome this problem, we could add a new opcode to allow the output script of $\mathtt{X}_{\mathsf{B}}\langle n\rangle$ to access the parameter in the redeeming transaction, so to verify that it is indeed $n+1$ as intended. Similarly, to check that we have $2\ddot{\mathrm{B}}$ more in the new transaction, an opcode could provide the value of the new output. The same goal could be achieved by adapting the techniques used in [24,25] to realize *covenants*.

6 Conclusions

We have investigated linguistic primitives to renegotiate BitML contracts, and their implementation on standard Bitcoin. More expressive primitives could be devised by relaxing this constraint, e.g. assuming the extended UTXO model [14].

The existing verification technique for BitML [12] is based on a sound and complete abstraction of the state space of contracts. Since this abstraction is finite-state, it can be model-checked to verify the required properties. The same technique can be directly applied to BitML contracts featuring renegotiation (but without recursion), since the abstraction would remain finite. Instead, the same abstraction on recursive contracts would lead to infinitely many states. Even if we could exploit the fact that Bitcoin uses 32-bit integers to make the state space finite, it would still be too large for verification to be practical.

If we assume that integers are unbounded, and that participants always accept renegotiations, the extension of BitML presented in Section 2 can simulate a counter machine, so making BitML Turing-complete. Hence, verification cannot be sound and complete. Alternative techniques to model checking (e.g., type-based approaches [15]) could be used to analyse relevant contract properties.

Acknowledgements Massimo Bartoletti is partially supported by Aut. Reg. of Sardinia projects *Sardcoin* and *Smart collaborative engineering*. Maurizio Murgia and Roberto Zunino are partially supported by MIUR PON *Distributed Ledgers for Secure Open Communities*.

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). https://doi.org/10.1007/978-3-662-44774-1_8
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P. pp. 443–458 (2014). https://doi.org/10.1109/SP.2014.35, first appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM **59**(4), 76–84 (2016). https://doi.org/10.1145/2896386
4. 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). https://doi.org/10.1007/978-3-662-54455-6_8
5. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: SoK: unraveling Bitcoin smart contracts. In: POST. LNCS, vol. 10804, pp. 217–242. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6
6. Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., Zunino, R.: Developing secure Bitcoin contracts with BitML. In: ESEC/FSE (2019). https://doi.org/https://doi.org/10.1145/3338906.3341173
7. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Financial Cryptography and Data Security. LNCS, vol. 10957. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6
8. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS. LNCS, vol. 9879, pp. 261–280. Springer (2016). https://doi.org/10.1007/978-3-319-45741-3_14
9. Bartoletti, M., Murgia, M., Zunino, R.: Renegotiation and recursion in Bitcoin contracts. CoRR **abs/2003.00296** (2020)
10. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0
11. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. In: ACM CCS (2018). https://doi.org/10.1145/3243734.3243795
12. Bartoletti, M., Zunino, R.: Verifying liquidity of bitcoin contracts. In: POST. LNCS, vol. 11426. Springer (2019)
13. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014). https://doi.org/10.1007/978-3-662-44381-1_24
14. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: Workshop on Trusted Smart Contracts (2020)
15. Das, A., Balzer, S., Hoffmann, J., Pfenning, F.: Resource-aware session types for digital contracts. CoRR **abs/1902.06056** (2019)
16. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: Sok: Off the chain transactions. IACR Cryptology ePrint Archive **2019**, 360 (2019)
17. Jones, S.L.P., Eber, J., Seward, J.: Composing contracts: an adventure in financial engineering, functional pearl. In: International Conference on Functional Programming (ICFP). pp. 280–292 (2000). https://doi.org/10.1145/351240.351267
18. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014). https://doi.org/10.1145/2660267.2660380
19. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: ACM CCS. pp. 418–429 (2016). https://doi.org/10.1145/2976749.2978424

20. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015). https://doi.org/10.1145/2810103.2813712
21. Kumaresan, R., Vaikuntanathan, V., Vasudevan, P.N.: Improvements to secure computation with penalties. In: ACM CCS. pp. 406–417 (2016). https://doi.org/10.1145/2976749.2978421
22. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS. pp. 254–269 (2016). https://doi.org/10.1145/2976749.2978309
23. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops. pp. 4–13 (2017). https://doi.org/10.1109/EuroSPW.2017.44
24. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography Workshops. LNCS, vol. 9604, pp. 126–141. Springer (2016). https://doi.org/10.1007/978-3-662-53357-4_9
25. O'Connor, R., Piekarska, M.: Enhancing Bitcoin transactions with covenants. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0_12