BitML: a calculus for Bitcoin smart contracts

Massimo Bartoletti
University of Cagliari
Cagliari, Italy
bart@unica.it

Roberto Zunino
University of Trento
Trento, Italy
roberto.zunino@unitn.it

***Abstract*—We propose a domain-specific language for smart contracts, which allows participants to transfer cryptocurrency according to agreed contract terms. We define a symbolic and a computational model for reasoning about their security. In the symbolic model, participants act according to the semantics of the domain-specific language. Instead, in the computational model they exchange bitstrings, and publish transactions on the Bitcoin blockchain. A compiler is provided to translate smart contracts into standard Bitcoin transactions. We prove the correctness of our compiler, showing that computational attacks to compiled smart contracts are also observable in the symbolic model.**

***Index Terms*—Bitcoin, smart contracts, process calculi**

I. Introduction

Cryptocurrencies like Bitcoin and Ethereum have revived the idea of *smart contracts*, agreements between untrusted parties that can be automatically enforced without a trusted intermediary [1]. These agreements define protocols which regulate cryptocurrency exchanges among participants. For instance, a smart contract can be a lottery which collects bets from a set of players, determines the winner in a fair manner, and then transfers the whole pot to the winner.

Disintermediation is made possible by the *blockchain*, a public, append-only record of transactions, and by the *consensus protocol* of the cryptocurrency [2]. The execution of smart contracts relies on the blockchain to log all the participants' moves; further, the underlying logic of transactions is exploited to enable all and only the moves permitted by the contract. The consensus protocol is used to consistently update the blockchain: economic incentives are provided to ensure that the nodes of the cryptocurrency network have the same view of the blockchain. In this way, the state of each contract (and consequently, the asset of each user) is uniquely determined by the sequence of its transactions on the blockchain.

Smart contracts have different incarnations, depending on the cryptocurrency on which they are based. In Ethereum, contracts are expressed as programs in a Turing-equivalent language. Any user can publish a contract on the blockchain. This makes the contract available to other users, who can then run it by calling its functions (concretely, by publishing suitable transactions on the blockchain). Such openness comes at the price of a wide attack surface: attackers may exploit vulnerabilities in the implementation of contracts, or may publish themselves Trojan-horse contracts with hidden vulnerabilities, to steal or tamper with the assets controlled by contracts. Indeed, a series of vulnerabilities in Ethereum contracts [3] have been exploited, causing money losses in the order of hundreds of millions of dollars [4]–[6].

Unlike Ethereum, Bitcoin does not provide a language for smart contracts: rather, they are described as cryptographic protocols where participants send/receive messages, verify signatures, and put/search transactions on the blockchain. Lotteries [7]–[10], Poker [11], contingent payments [12]–[14], and more general fair multi-party computations [15], [16] witness the variety of smart contracts supported by Bitcoin.

Describing smart contracts at this level of abstraction is complex and error-prone, for two reasons. First, they often rely on advanced features of Bitcoin, (e.g., transaction scripts, signature modifiers, temporal constraints), whose actual behaviour relies on low-level implementation details. Second, establishing the correctness of each smart contract requires to prove the computational security of its protocol. This task requires the skills of expert cryptographers, and even in this case it is a significant effort. By contrast, working in an high-level symbolic model would relieve smart contract programmers from (most of) this burden, since the much higher level of abstraction would allow security proofs to be carried out with automatic tools.

Contributions: We propose BitML, a domain-specific language for Bitcoin smart contracts. BitML is a process calculus, with primitives to stipulate contracts, and to exchange currency according to the contract terms, possibly involving temporal constraints. BitML can express most of the Bitcoin smart contracts proposed so far in the literature [17], e.g. escrow services, timed commitments, multi-player lotteries, etc.

We provide our calculus with a *symbolic semantics*, and a security model where participants use strategies to choose which moves to perform (among those allowed by the semantics). Each honest participant has her own strategy, while the dishonest ones are controlled by a single adversarial strategy. The adversary is based on the (symbolic) Dolev-Yao model: he schedules the participant moves, eavesdrops messages, and impersonates other participants. However, we prevent the adversary from dropping messages forever, coherently with the peer-to-peer nature of the Bitcoin communication network, where (proper) requests are always eventually processed. Since BitML features temporal constraints, our security model handles time. More specifically, we allow the adversary to let time pass, but only provided that the other participants are also willing to. In this way we rule out attacks where the adversary advances the time making a participant miss a deadline.

Our main contribution is a compiler which translates BitML contracts into standard Bitcoin transactions. Participants can

perform the actions of the contract by publishing the corresponding transactions on the blockchain. The main technical challenge is to guarantee the correctness of the compiler, i.e. that the symbolic execution of the contract matches the "computational" one performed on the Bitcoin network. This correspondence must hold also in the presence of computational adversaries: otherwise, attacks at the Bitcoin level could be unobservable at the level of the symbolic semantics.

To prove the correctness of the compiler, we start by defining a *computational model* of the honest participants and of the adversary. Both are equipped with computational strategies, which allow them to inspect the past run, and choose which actions to perform. In the computational model, actions can broadcast bitstrings, publish transactions on the blockchain, and make time pass. Compared to the symbolic level, the computational level extends the power of the adversary: he can arbitrarily manipulate bitstrings, being only subject to complexity bounds, and he can exploit any kind of valid Bitcoin transaction, beyond those obtained by the compiler.

Then, we introduce a relation between symbolic and computational runs, called *coherence*. Intuitively, coherence holds when both runs perform "morally" equivalent actions: e.g., the same amount of bitcoins is spent in both runs at the same time, for the same purpose. Defining coherence is technically demanding, because the symbolic model is far more abstract than the computational one; further, the computational adversary can perform actions with no symbolic counterpart.

To illustrate one of the subtleties in formalising coherence, suppose that a dishonest participant A owns v฿, represented both in the symbolic run (as a term $\langle \mathsf{A}, v \rangle$) and in the computational run (as a transaction T). Computationally, A can redeem T with a transaction T' which can not be represented symbolically (e.g., because it can not be produced by the compiler). In such case, a coherent symbolic step is to make $\langle \mathsf{A}, v \rangle$ disappear, as if v฿ were destroyed. In subsequent steps, coherence will not keep track of the descendants of T' in the symbolic run. This loss of information at the symbolic level could in principle allow for computational attacks without a symbolic counterpart. However, this is not the case, since computational attacks can always be adapted so to have a symbolic counterpart. Indeed, to attack honest participants, A has to stipulate contracts with them: this requires A to put a deposit, computationally represented as a transaction T''. Instead of obtaining T'' from T', which makes T'' untraceable at the symbolic level, A can perform symbolically-traceable actions to create from T a deposit T''' with the same value of T'', to be used in the computational attack. This adaptation is feasible, because untraceable computational actions do not allow the adversary to artificially increase his wealth. Hence, the value of T'' can not exceed the value of T, so making it possible to produce T''' with symbolic actions.

Leveraging on the definition of coherence, we establish the correctness of the compiler, as a *computational soundness* result [18]. Specifically, we prove that if honest participants execute contracts using symbolic strategies and compiler-generated transactions, then the computational runs resulting from their interaction with the adversary will have a symbolic counterpart (with overwhelming probability). Consequently, proofs of trace-based security properties carried out in the symbolic model can be lifted *for free* to the computational model. This result is crucial, since it enables the development of analysis and verification techniques *at the symbolic level*, which would be much more burdensome to obtain at the (far more concrete) computational level.

II. A CALCULUS FOR BITCOIN CONTRACTS

In BitML, contracts allow participants to interact according to the following workflow:

1) A participant broadcasts a *contract advertisement* $\{G\}C$. The component C is the contract, which specifies how funds can be transferred among participants; G is a set of *preconditions* to its execution. Roughly, G requires participants to deposit some ฿, either upfront or during the contract execution, and to commit to some secrets.
2) Participants can then choose whether to accept the advertisement, or not. When all the involved participants have accepted $\{G\}C$, satisfying its preconditions, the contract C becomes stipulated. Then, participants can transfer the deposited funds by acting as prescribed by C.

We assume a set Part of *participants*, ranged over by $\mathsf{A}, \mathsf{B}, \ldots$, and we denote with $\mathsf{Hon} \subseteq \mathsf{Part}$ a non-empty set of *honest* participants. We also assume a set of names, of two kinds: $x, y, \ldots$ denote *deposits* of ฿, while $a, b, \ldots$ denote *secrets*. We denote with $\tilde{x}$ a finite sequence of deposit names, and we adopt a similar notation for sequences of other kinds.

The preconditions in G have three possible forms:

- $\mathsf{A} : ! \, v \, @ \, x$ ("persistent deposit") requires A to own v฿ in a deposit x, and to spend it for stipulating the contract.
- $\mathsf{A} : ? \, v \, @ \, x$ ("volatile deposit") only requires A to authorize the spending of x during the execution of C. Since x is not paid upfront, there is no guarantee that v฿ will be available when C demands x, as A can spend it for other purposes. Intuitively, volatile deposits $\mathsf{A} : ? \, v \, @ \, x$ give A the option of contributing v฿ during the contract execution, but there is no obligation for A to do so.
- $\mathsf{A} : \mathtt{secret} \, a$ requires A to generates a random nonce a, and commit to it by publishing its hash $H(a)$ before C starts. During the execution of C, A can choose whether to reveal a to the other participants.

Definition 1 (Contract preconditions).

$G ::=$	contract precondition
$\mathsf{A} : ? \, v \, @ \, x$	volatile deposit of v฿, expected from A
$\mid \mathsf{A} : ! \, v \, @ \, x$	persistent deposit of v฿, expected from A
$\mid \mathsf{A} : \mathtt{secret} \, a$	committed secret by A
$\mid G \mid G$	composition

Once C is stipulated, it starts its execution with a *balance*, initially set to the sum of the persistent deposits in its advertisement. The execution of C will affect this balance, when participants deposit/withdraw funds to/from the contract.

We formalise contracts using a process calculus. A contract C is a *choice* among branches. Intuitively, a branch D performs an action, and possibly proceeds with a continuation C'. The action $\texttt{put}\,\tilde{x}\,\&\,\texttt{reveal}\,\tilde{a}\,\texttt{if}\,p$ atomically performs the following: (i) spend all the volatile deposits $\tilde{x}$, adding their values to the current balance; (ii) check that all the secrets $\tilde{a}$ have been revealed, and that they 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 be equal to the current balance.

The prefix $\texttt{withdraw}\ \mathsf{A}$ transfers the whole balance to A (to transfer only a part of it, one can perform a `split`).

Note that, 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 $\mathsf{A} : D$, which requires the authorization of A, and the decoration $\texttt{after}\,t : D$, which requires to wait until time t.

Definition 2 (Contracts). We define the syntax of contracts in Figure 1. We denote with 0 the empty sum. We abbreviate $\texttt{put}\,\tilde{x}\,\&\,\texttt{reveal}\,\tilde{a}\,\texttt{if}\,p$ as: (i) $\texttt{put}\,\tilde{x}$ when $\tilde{a}$ is empty and p is *true*, (ii) $\texttt{reveal}\,\tilde{a}\,\texttt{if}\,p$ when $\tilde{x}$ is empty, (iii) τ when $\tilde{x}$ and $\tilde{a}$ are empty and p is *true*, and (iv) we omit "$\texttt{if}\,p$" when the predicate p is true. In guarded contracts, we assume that the order of decorations is immaterial, e.g., we consider $\texttt{after}\,t : \mathsf{A} : \mathsf{B} : D$ equivalent to $\mathsf{B} : \mathsf{A} : \texttt{after}\,t : D$.

Definition 3 (Contract advertisement). A contract advertisement is a term $\{G\}C$ such that the following conditions hold: (i) all the names in G are distinct; (ii) all the names in C must occur in G; (iii) in $\texttt{put}\,\tilde{x}\,\&\,\texttt{reveal}\,\tilde{a}\,\texttt{if}\,p$, all the names are distinct; further, the names in p must occur in $\tilde{a}$; (iv) each participant occurring in $\{G\}C$ has a persistent deposit in G.

The last condition is not restrictive in practice: we can craft a contract that allows a participant A to deposit just a small fraction of bitcoin, and then immediately transfer it back to A through a `split`. In the following examples, we do not show these dummy deposits in the contract preconditions.

Example 1 (Escrow). A buyer A wants to buy an item from a seller B, but they do not trust each other. So, they would like to use a contract to ensure that B will get paid if and only if A gets her item. We would like to guarantee that, even if either A or B are dishonest, exactly one of them will be able to redeem the money: if they disagree, an arbiter C will decide who gets the money.

We model this contract in BitML as follows. The precondition $G = \mathsf{A} : !\, v\, @\, x$ requires A to provide a persistent deposit of v฿. The contract C is a choice among four branches:

$$\begin{aligned} C \;=\; & \mathsf{A} : \texttt{withdraw}\ \mathsf{B} \;+\; \mathsf{B} : \texttt{withdraw}\ \mathsf{A} \\ & + \mathsf{C} : \texttt{withdraw}\ \mathsf{A} \;+\; \mathsf{C} : \texttt{withdraw}\ \mathsf{B} \end{aligned}$$

In the first branch, $\mathsf{A} : \texttt{withdraw}\ \mathsf{B}$, participant B can redeem v฿ if A provides her authorization. The second branch is dual, and can be used e.g., if the seller wants to refund the buyer for a damaged item. The last two branches are used if neither A nor B give their authorizations: in this case, the arbiter C chooses whether to authorize A or B to redeem the deposit.

We also propose a variant of the contract where C can issue a *partial* refund of ζv฿ to A, and of $(1-\zeta)v$฿ to B (similarly to [19], [20]). The possible values of ζ are given by a finite set Z in the range $[0,1]$. We model the new contract as:

$$\begin{aligned} C' &= \mathsf{A} : \texttt{withdraw}\ \mathsf{B} + \mathsf{B} : \texttt{withdraw}\ \mathsf{A} + \textstyle\sum_{\zeta\in Z} \mathsf{C} : D_\zeta \\ D_\zeta &= \texttt{split}\ \big(\zeta v \to \texttt{withdraw}\ \mathsf{A} \mid (1-\zeta)v \to \texttt{withdraw}\ \mathsf{B}\big) \end{aligned}$$

The case of full refunds is obtained with $Z = \{0,1\}$. If $Z = \{0, 1/2, 1\}$, C can also choose to refund $v/2$฿ to both. ◇

Example 2 (Intermediated payment). Assume that A wants to send an indirect payment of v_{C}฿ to C, routing it through an intermediary B who can choose whether to authorize the payment, in this case retaining a fee v_{B}, or not. Since A does not trust B, she wants to use a contract to guarantee that: (i) if B is honest, then v_{C}฿ are transferred to C; (ii) if B is *not* honest, then A does not lose money.

In BitML, we use $G = \mathsf{A} : !\, v_{\mathsf{B}} + v_{\mathsf{C}}\, @\, x$ as precondition, and the following contract:

$$\begin{aligned} C \;=\; & \mathsf{B} : \texttt{split}\ \big(v_{\mathsf{B}} \to \texttt{withdraw}\ \mathsf{B} \mid v_{\mathsf{C}} \to \texttt{withdraw}\ \mathsf{C}\big) \\ & +\ \texttt{after}\,t : \texttt{withdraw}\ \mathsf{A} \end{aligned}$$

The first branch can only be taken if B authorizes the payment: in this case, B gets his fee, and C gets his payment. If B denies his authorization, after time t, A can redeem her deposit. ◇

Example 3 (Timed commitment). Assume that A wants to choose a secret a, and reveal it after some time t — guaranteeing that the revealed value is the chosen secret. This can be obtained through a *timed commitment* protocol [15], [21]–[23]. As in [15], we force A to pay to B a penalty of v฿ if A does not reveal the secret within the t. In BitML, we use the precondition $G = \mathsf{A} : !\, v\, @\, x \mid \mathsf{A} : \texttt{secret}\, a$, and:

$$C \;=\; \big(\texttt{reveal}\, a.\, \texttt{withdraw}\ \mathsf{A}\big) \;+\; \big(\texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\big)$$

Only A can choose the first branch, by revealing a. After that, anyone can further reduce the contract, and transfer v฿ to A. Only after time t, B can choose the second branch, and collect A's penalty. Before the deadline, A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). If A reveals a after time t, a race condition occurs: in such case, the first one who fires the `withdraw` gets the money.

We can also model *mutual* timed commitment as follows:

$$\begin{aligned} G &= \mathsf{A} : !\, v\, @\, x \mid \mathsf{A} : \texttt{secret}\, a \mid \mathsf{B} : !\, v\, @\, y \mid \mathsf{B} : \texttt{secret}\, b \\ C &= \texttt{reveal}\, a.\, C' \;+\; \big(\texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\big) \\ C' &= \texttt{reveal}\, b.\, C'' + \big(\texttt{after}\, t : \texttt{withdraw}\ \mathsf{A}\big) \\ C'' &= \texttt{split}\ \big(v \to \texttt{withdraw}\ \mathsf{A} \mid v \to \texttt{withdraw}\ \mathsf{B}\big) \end{aligned}$$

The contract C can reduce to C' if A reveals a, otherwise (after t) B can redeem $2v$฿. In C', if B reveals b, then both participants can redeem their deposits, running C''. Otherwise, A can redeem $2v$฿. ◇

$C ::= \sum_{i\in I} D_i$	contract	$p ::=$	predicate	$E ::=$	arithmetic expression
$D ::=$	guarded contract	true	truth	N	32-bit constant
$\mathtt{put}\ \boldsymbol{x}\ \&\ \mathtt{reveal}\ \boldsymbol{a}\ \mathtt{if}\ p.\, C$	collect deposits $\boldsymbol{x}$ and secrets $\boldsymbol{a}$	$\mid p \wedge p$	conjunction	$\mid \lvert a \rvert$	length of a secret
$\mid \mathtt{withdraw}\ \mathsf{A}$	transfer the balance to A	$\mid \neg p$	negation	$\mid E + E$	addition
$\mid \mathtt{split}\ \boldsymbol{v} \to \boldsymbol{C}$	split the balance ($\lvert \boldsymbol{v} \rvert = \lvert \boldsymbol{C} \rvert$)	$\mid E = E$	equality	$\mid E - E$	subtraction
$\mid \mathsf{A} : D$	wait for A's authorization	$\mid E < E$	less than		
$\mid \mathtt{after}\, t : D$	wait until time t				

Fig. 1: Syntax of BitML contracts.

Example 4 (Two-players lottery). A multiparty lottery is a protocol where n players put their bets in a pot, and a winner — fairly chosen among the players — redeems the whole pot. Various contracts for multiparty lotteries on Bitcoin have been proposed in [8]–[10], [15], [24], [25].

We model the lottery in [15], for two players A and B who bet 1฿ each. The contract preconditions are the following:

$$\begin{aligned} G \ = {} & \mathsf{A} : !\, 2 \mathbin{@} y_{\mathsf{A}} \mid \mathsf{A} : ?\, 1 \mathbin{@} x_{\mathsf{A}} \mid \mathsf{A} : \mathtt{secret}\, a \\ & \mid \mathsf{B} : !\, 2 \mathbin{@} y_{\mathsf{B}} \mid \mathsf{B} : ?\, 1 \mathbin{@} x_{\mathsf{B}} \mid \mathsf{B} : \mathtt{secret}\, b \end{aligned}$$

where a and b are committed secrets, y_{A}, y_{B}, are collaterals (used as compensations in case of dishonest behaviour), and x_{A}, x_{B} are the bets. The contract C is the following:

$$\begin{aligned} & \mathtt{put}\, x_{\mathsf{A}} x_{\mathsf{B}}.\, (\mathtt{split} \\ & \quad 2 \to \mathtt{reveal}\, b\, \mathtt{if}\, 0 \le |b| \le 1.\, \mathtt{withdraw}\ \mathsf{B} \\ & \qquad + \mathtt{after}\, t' : \mathtt{withdraw}\ \mathsf{A} \\ & \quad \mid 2 \to \mathtt{reveal}\, a\, \mathtt{if}\, 0 \le |a| \le 1.\, \mathtt{withdraw}\ \mathsf{A} \\ & \qquad + \mathtt{after}\, t' : \mathtt{withdraw}\ \mathsf{B} \\ & \quad \mid 2 \to \mathtt{reveal}\, a b\, \mathtt{if}\, |a| = |b|.\, \mathtt{withdraw}\ \mathsf{A} \\ & \qquad + \mathtt{reveal}\, a b\, \mathtt{if}\, |a| \neq |b|.\, \mathtt{withdraw}\ \mathsf{B}\,) \\ & + \mathtt{after}\, t : \mathtt{split}\ (2 \to \mathtt{withdraw}\ \mathsf{A} \mid 2 \to \mathtt{withdraw}\ \mathsf{B}\,) \end{aligned}$$

The players can put their bets by performing `put`; if they do not place their bets within t, each player can use the `after` branch to get their collateral back. After the bets are placed, the balance is split in three parts. Player B must reveal b before the deadline t'; otherwise, A can redeem B's collateral (note that this is a timed commitment, as in Example 3). Similarly, A must reveal a. To fairly determine the winner, we further require that the length of secrets is either 0 or 1 (after compiling to Bitcoin, the actual lengths are $\eta + 0$ and $\eta + 1$ so to avoid trivial preimage attacks). In this way, the third part of the `split` can compute the winner: A if the secrets have the same length, B otherwise. ◇

We now define a reduction semantics of contracts. It is organised in two layers: a bottom layer, taking the form of an LTS between (untimed) *configurations*, and a top layer, in the form of a timed LTS between *timed configurations*.

An (untimed) configuration Γ is a parallel composition of:

- *contract advertisements* $\{G\}C$, representing a contract which has been proposed, but not stipulated yet.
- *active contracts* $\langle C, v\rangle_x$, representing a stipulated contract, holding a current balance of v฿. The name x uniquely identifies the active contract.
- *personal deposits* $\langle \mathsf{A}, v\rangle_x$, representing a fund of v฿ owned by A, and with unique name x.
- *authorizations* $\mathsf{A}[\chi]$, representing the consent of A to perform some operation χ.
- *committed secrets* $\{\mathsf{A} : a \# N\}$, i.e. A has committed a random nonce a, by broadcasting its hash $H(a)$. The length of a, which is secret as well, is determined by N.
- *revealed secrets* $\mathsf{A}[a \# N]$, representing the fact that A has revealed her secret a (hence, its length N).

Definition 4 (Configurations). We define configurations $\Gamma, \Delta, \ldots$ through the syntax in Figure 2, where we stipulate that in a configuration there are no duplicate authorizations. We assume that $(|, 0)$ is a commutative monoid. Indexed parallel compositions are denoted with $\|$. We say that Γ is *initial* when it contains only terms of the form $\langle \mathsf{A}, v\rangle_x$.

Definition 5 (LTS of untimed configurations). The LTS of configurations is defined in Figure 3 (advertisements and stipulation), Figure 4 (contracts), and Figure 8 (deposits). The latter is deferred to Section A, as for the semantics of predicates (Definition 24). The function cv from labels α to sets of names is defined as: $cv(\mathit{put}(\boldsymbol{x}, \boldsymbol{a}, y)) = cv(\mathit{withdraw}(\mathsf{A}, v, y)) = cv(\mathit{split}(y)) = \{y\}$, otherwise $cv(\alpha)$ is empty. ◇

When a participant A owns a deposit $\langle \mathsf{A}, v\rangle_x$, she can employ that amount for several operations. For instance, she can divide the deposit into two smaller deposits, or join it with another deposit of hers to form a larger one. The deposit can also be transferred to another participant, or used to stipulate a contract. Before such operations can be performed, A must authorize it (at the Bitcoin level, this happens through suitable signatures of A). For this reason, we model both the authorization step and the actual operation using two distinct LTS moves, formalized by two separate semantics rules. For instance, rule [Dep-AuthJoin] authorizes the join of two deposits:

$$\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \to \langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \mathsf{A}[\chi_x] \mid \Gamma$$

where $\chi_x = x, y \triangleright \langle \mathsf{A}, v + v'\rangle$ authorizes to spend x. After we also obtain the dual authorization χ_y, rule [Dep-Join] performs the actual join. Given $\Gamma = \mathsf{A}[\chi_x] \mid \mathsf{A}[\chi_y] \mid \Gamma'$, we infer:

$$\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \to \langle \mathsf{A}, v + v'\rangle_z \mid \Gamma$$

Rules for contracts are better explained through an example.

$\Gamma, \Delta ::=$	untimed configuration		
0	empty	$\chi ::=$	authorization to …
$\mid \{G\}C$	contract advertisement	$x \rhd \{G\}C$	accept an advertised contracts $\{G\}C$
$\mid \langle C, v\rangle_x$	an active contract containing v฿	$\mid x \rhd D$	take branch D
$\mid \langle \mathsf{A}, v\rangle_x$	a deposit of v฿ redeemable by A	$\mid x, y \rhd \langle \mathsf{A}, v\rangle$	join deposit x with y into a deposit for A
$\mid \mathsf{A}[\chi]$	authorization of A to perform χ	$\mid x \rhd \langle \mathsf{A}, v\rangle, \langle \mathsf{A}, v'\rangle$	divide a deposit x in two deposits for A
$\mid \{\mathsf{A} : a\#N\}$	committed secret of A ($N \in \mathbb{N} \cup \{\bot\}$)	$\mid x \rhd \mathsf{B}$	donate deposit x to B
$\mid \mathsf{A}[a\#N]$	revealed secret of A ($N \in \mathbb{N}$)	$\mid \boldsymbol{x}, i \rhd y$	destroy i-th deposits in $\boldsymbol{x}$ through y
$\mid \Gamma \mid \Delta$	parallel composition		

Fig. 2: Configurations.

$$\frac{\{G\}C \text{ contains at least one participant in } \mathsf{Hon} \qquad \Gamma \text{ contains } \langle \mathsf{A}_i, v_i\rangle_{x_i} \text{ for all } \mathsf{A}_i : !\, v_i \,@\, x_i \text{ in } \{G\}C \\ a \text{ fresh, for each } \mathsf{A} : \mathtt{secret}\, a \text{ in } G \qquad \Gamma \text{ contains } \langle \mathsf{A}_i, v_i\rangle_{x_i} \text{ for all } \mathsf{A}_i : ?\, v_i \,@\, x_i \text{ in } \{G\}C}{\Gamma \xrightarrow{advertise(\{G\}C)} \{G\}C \mid \Gamma} \text{[C-Advertise]}$$

$$\frac{a_1 \cdots a_k \text{ secrets of } \mathsf{A} \text{ in } G \qquad \forall i \in 1..k : N_i \in \begin{cases} \mathbb{N} & \text{if } \mathsf{A} \in \mathsf{Hon} \\ \mathbb{N} \cup \{\bot\} & \text{otherwise} \end{cases} \\ \forall i \in 1..k : \nexists N : \{\mathsf{A} : a_i\#N\} \in \Gamma \qquad \Delta = \{\mathsf{A} : a_1\#N_1\} \mid \cdots \mid \{\mathsf{A} : a_k\#N_k\}}{\{G\}C \mid \Gamma \xrightarrow{\mathsf{A}:\{G\}C,\Delta} \{G\}C \mid \Gamma \mid \Delta} \text{[C-AuthCommit]}$$

$$\frac{\forall \mathsf{B} : \forall a \text{ secret of } \mathsf{B} \text{ in } G : \exists N : \big(\{\mathsf{B} : a\#N\} \in \Gamma \text{ or } \mathsf{B}[a\#N] \in \Gamma\big) \qquad G \text{ contains a deposit } x \text{ of } \mathsf{A}}{\{G\}C \mid \Gamma \xrightarrow{\mathsf{A}:\{G\}C,x} \{G\}C \mid \Gamma \mid \mathsf{A}[x \rhd \{G\}C]} \text{[C-AuthInit]}$$

$$\frac{G = \big(\|_{i\in I} \mathsf{A}_i : ?\, v_i \,@\, x_i\big) \mid \big(\|_{i\in J} \mathsf{B}_i : !\, v'_i \,@\, y_i\big) \mid \big(\|_{i\in K} \mathsf{C}_i : \mathtt{secret}\, a_i\big) \qquad x \text{ fresh} \\ \Gamma = \big(\|_{i\in I} \mathsf{A}_i[x_i \rhd \{G\}C]\big) \mid \big(\|_{i\in J} \mathsf{B}_i[y_i \rhd \{G\}C]\big) \mid \Gamma'}{\{G\}C \mid \Gamma \mid \big(\|_{i\in J} \langle \mathsf{B}_i, v'_i\rangle_{y_i}\big) \xrightarrow{init(G,C)} \langle C, \textstyle\sum_{i\in J} v'_i\rangle_x \mid \Gamma} \text{[C-Init]}$$

Fig. 3: Semantics of untimed configurations: rules for advertisement and stipulation.

$$\frac{\boldsymbol{v} = v_1 \cdots v_k \qquad \boldsymbol{C} = C_1 \cdots C_k \qquad \textstyle\sum_{i=1}^k v_k = v' \qquad x_1 \cdots x_k \text{ fresh}}{\langle \mathtt{split}\, \boldsymbol{v} \to \boldsymbol{C}, v'\rangle_y \mid \Gamma \xrightarrow{split(y)} \big(\|_{i=1}^k \langle C_i, v_i\rangle_{x_i}\big) \mid \Gamma} \text{[C-Split]}$$

$$\frac{N \neq \bot}{\{\mathsf{A} : a\#N\} \xrightarrow{\mathsf{A}:a} \mathsf{A}[a\#N]} \text{[C-AuthRev]} \qquad \frac{\boldsymbol{x} = x_1 \cdots x_m \qquad \Gamma = \|_{i=1}^m \langle \mathsf{A}_i, v_i\rangle_{x_i} \qquad z \text{ fresh} \\ \boldsymbol{a} = a_1 \cdots a_n \qquad \Delta = \|_{i=1}^n \mathsf{B}_i[a_i\#N_i] \qquad [\![p]\!]_\Delta = true}{\langle \mathtt{put}\, \boldsymbol{x}\, \&\, \mathtt{reveal}\, \boldsymbol{a}\, \mathtt{if}\, p.\, C, v\rangle_y \mid \Gamma \mid \Delta \mid \Gamma' \xrightarrow{put(\boldsymbol{x},\boldsymbol{a},y)} \langle C, v + \textstyle\sum_{i=1}^m v_i\rangle_z \mid \Delta \mid \Gamma'} \text{[C-PutRev]}$$

$$\frac{x \text{ fresh}}{\langle \mathtt{withdraw}\, \mathsf{A}, v\rangle_y \mid \Gamma \xrightarrow{withdraw(\mathsf{A},v,y)} \langle \mathsf{A}, v\rangle_x \mid \Gamma} \text{[C-Withdraw]} \qquad \frac{D \equiv \mathsf{A} : D'}{\langle D + C, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,D} \langle D + C, v\rangle_x \mid \mathsf{A}[x \rhd D] \mid \Gamma} \text{[C-AuthControl]}$$

$$\frac{D \equiv \mathsf{A}_1 : \cdots : \mathsf{A}_k : \mathtt{after}\, t_1 : \cdots : \mathtt{after}\, t_m : D' \qquad D' \not\equiv \mathsf{A} : \cdots \\ \langle D', v\rangle_x \mid \|_{i=1}^k \mathsf{A}_i[x \rhd D] \mid \Gamma \xrightarrow{\alpha} \Gamma' \qquad x \in cv(\alpha) \qquad D' \not\equiv \mathtt{after}\, t : \cdots}{\langle D + C, v\rangle_x \mid \|_{i=1}^k \mathsf{A}_i[x \rhd D] \mid \Gamma \xrightarrow{\alpha} \Gamma'} \text{[C-Control]}$$

Fig. 4: Semantics of untimed configurations: rules for contracts.

$$\frac{\Gamma \xrightarrow{\alpha} \Gamma'}{(\Gamma, t) \xrightarrow{\alpha} (\Gamma', t)} \text{[Action]} \qquad \frac{\delta > 0}{(\Gamma, t) \xrightarrow{\delta} (\Gamma, t + \delta)} \text{[Delay]}$$

$$\frac{D \equiv \mathtt{after}\, t_1 : \cdots : \mathtt{after}\, t_m : D' \qquad D' \not\equiv \mathtt{after}\, t' : \cdots \\ \langle D, v\rangle_x \mid \Gamma \xrightarrow{\alpha} \Gamma' \qquad x \in cv(\alpha) \qquad t \geq t_1, \ldots, t_m}{\big(\langle D + C, v\rangle_x \mid \Gamma, t\big) \xrightarrow{\alpha} (\Gamma', t)} \text{[Timeout]}$$

Fig. 5: Semantics of timed configurations.

Example 5. Recall the timed commitment in Example 3. Assume that $\mathsf{A} \in \mathsf{Hon}$, and let $\Gamma = \langle \mathsf{A}, v \rangle_x$. A possible (untimed) computation, where A reveals her secret and then redeems the deposit, is the following:

$$\begin{aligned}
\Gamma &\to \Gamma \mid \{G\}C && (1)\\
&\to \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} && (2)\\
&\to \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[x \rhd \{G\}C] && (3)\\
&\to \langle C, v \rangle_{x_1} \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[x \rhd \{G\}C] \;=\; \Gamma_1 && (4)\\
&\to \langle C, v \rangle_{x_1} \mid \mathsf{A}[a\#N] \mid \mathsf{A}[x \rhd \{G\}C] && (5)\\
&\to \langle \texttt{withdraw}\ \mathsf{A}, v \rangle_{x_2} \mid \mathsf{A}[a\#N] \mid \mathsf{A}[x \rhd \{G\}C] && (6)\\
&\to \langle \mathsf{A}, v \rangle_{x_3} \mid \mathsf{A}[a\#N] \mid \mathsf{A}[x \rhd \{G\}C] && (7)
\end{aligned}$$

Step (1) advertises $\{G\}C$, by rule [C-Advertise]. This move is possible because A is honest, and her deposit is in Γ. At step (2), A commits to a secret, by rule [C-AuthCommit]. This is possible because there are no previous commitments of a in Γ; the length N is a natural, since A is honest. At step (3), A gives her authorization to stipulate C, by rule [C-AuthInit]. This is possible since the configuration contains A's deposit and committed secret. At step (4) the contract C becomes stipulated, by rule [C-Init]. Stipulation is possible because A authorized it, and made available her persistent deposit x. After the move, the v₿ in x are transferred to the contract. At step (5), A reveals her secret by rule [C-AuthRev], which verifies that $N \neq \bot$. After that, the action $\texttt{reveal}\, a$ is performed at step (6), by rule [C-PutRev], followed by [C-Control] with $k = m = 0$. Finally, A withdraws v₿ at step (7) by rule [C-Withdraw], obtaining a fresh deposit x_3. ◇

We discuss a few subtleties in the rules. First, [C-Advertise] requires as a side condition that at least one of the participants involved in each stipulation is honest (one of the weakest assumptions in cryptographic protocols). The same effect of running contract among dishonest participants can still be obtained by redistributing funds through the rules for deposits. Hence, this side condition does not affect the power of the adversary. A further motivation for the side condition is that the correctness of our compiler will rely on this assumption.

Rule [C-AuthCommit] allows dishonest participants to choose an "invalid" length $\bot$ for their committed secrets. This reflects the fact that, in the computational model, A commits to a secret by broadcasting a bitstring, meant to be the hash of the secret. If A is dishonest, she could instead broadcast an arbitrary bitstring w, preventing herself later on to reveal a preimage of w. Similarly, the length $\bot$ prevents the reveal action in the symbolic model.

Definition 6 (LTS of timed configurations). Timed configurations are pairs (Γ, t), where $t \in \mathbb{N}$ is the global time. The LTS between timed configurations is defined in Figure 5, where labels α are either untimed labels, or delays $\delta \in \mathbb{N}$. ◇

Example 6. Recall the configuration Γ_1 from Example 5. Starting from $(\Gamma_1, 0)$ we can have an alternative timed computation, where B waits until $t' > t$ to redeem A's deposit:

$$\begin{aligned}
(\Gamma_1, 0) &\to (\Gamma_1, t') && (8)\\
&\to (\langle \mathsf{B}, v \rangle_y \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[x \rhd \{G\}C], t') && (9)
\end{aligned}$$

Step (8) lets the time pass, by rule [Delay]. At step (9), B fires the prefix $\texttt{withdraw}\ \mathsf{B}$ within the $\texttt{after}$, and in this way he collects v₿ from A's deposit. This is obtained by using rule [C-Withdraw] in the premise of [C-Timeout]. ◇

III. Symbolic strategies and adversaries

A symbolic strategy Σ^s is a PPTIME algorithm which allows participants to select which action(s) to perform, among those permitted by the semantics. We distinguish between two kinds of strategies: those for honest participants, and that for the adversary Adv (who also controls the dishonest participants). Strategies can inspect the past execution.

Definition 7 (Symbolic runs). A symbolic run R^s is a (possibly infinite) sequence $(\Gamma_0, t_0) \xrightarrow{\alpha_0} (\Gamma_1, t_1) \xrightarrow{\alpha_1} \cdots$ where Γ_0 is initial, and $t_0 = 0$. If R^s is finite, we write Γ_{R^s} for its last untimed configuration, and δ_{R^s} for the last global time. We write $R^s \xrightarrow{\alpha} \dot{R}^s$ when $\dot{R}^s$ extends R^s with the transition $(\Gamma_{R^s}, \delta_{R^s}) \xrightarrow{\alpha} (\Gamma_{\dot{R}^s}, \delta_{\dot{R}^s})$. ◇

The strategy of a participant can inspect the past run, except for the (lengths of the) committed secrets which have not been revealed yet. The *stripping* of a symbolic run censors this information. Instead, revealed secrets are not censored.

Definition 8 (Stripping of symbolic runs). The stripping of a symbolic run, denoted $\mathit{strip}(R^s)$, is the sequence obtained from R^s by replacing each committed secret $\{\mathsf{A} : a\#N\}$ with $\{\mathsf{A} : a\#\bot\}$, and each label $\mathsf{A} : \{G\}C, \Delta$ with $\mathsf{A} : \{G\}C, 0$. We accordingly define label stripping. ◇

A strategy receives as input a stripped run R^s_*. Further, the strategy of A has access to an infinite sequence $r_\mathsf{A} \in \{0,1\}^\omega$ of independent and uniformly random bits. Instead of modelling the access to r_A through an oracle, we simply pass r_A as input to the strategy.

Definition 9 (Randomness sources). A randomness source is a function r from participants $\mathsf{A} \in \mathsf{Hon} \cup \{\mathsf{Adv}\}$ to infinite bitstrings $r(\mathsf{A}) \in \{0,1\}^\omega$. We usually write r_A for $r(\mathsf{A})$. ◇

Definition 10 (Symbolic participant strategies). The symbolic strategy of a participant $\mathsf{A} \in \mathsf{Hon}$ is a PPTIME algorithm $\Sigma^s_\mathsf{A}(R^s_*, r_\mathsf{A})$, taking as input a stripped symbolic run R^s_* and a random sequence r_A. The output is a finite sequence of α-moves such that the following constraints hold:

1) if $\alpha \in \Sigma^s_\mathsf{A}(\mathit{strip}(R^s), r_\mathsf{A})$, then $R^s \xrightarrow{\alpha}$;
2) if $\mathsf{B} : \cdots \in \Sigma^s_\mathsf{A}(\mathit{strip}(R^s), r_\mathsf{A})$, then $\mathsf{B} = \mathsf{A}$;
3) if $\mathsf{A} : \{G\}C, \Delta$ and $\mathsf{A} : \{G\}C, \Delta'$ in $\Sigma^s_\mathsf{A}(\mathit{strip}(R^s), r_\mathsf{A})$, then $\Delta = \Delta'$;
4) if $\alpha \in \Sigma^s_\mathsf{A}(\mathit{strip}(R^s), r_\mathsf{A})$ and $R^s \xrightarrow{\alpha_1} \dot{R}^s \xrightarrow{\alpha}$, then $\alpha \in \Sigma^s_\mathsf{A}(\mathit{strip}(\dot{R}^s), r_\mathsf{A})$. ◇

The constraints in Definition 10 are needed to rule out ill-formed strategies. (1) requires that Σ^s_A only chooses moves enabled by the semantics; (2) states that Σ^s_A cannot choose moves of B; (3) guarantees that the lengths of secrets are chosen coherently (i.e., A cannot choose different lengths for the same secret); (4) requires the strategy to be *persistent*: if

on a run Σ^s_{A} chooses α, and α is not taken as the next step in the run (e.g., because some other participant acts earlier), then Σ^s_{A} must still choose α after that step, if still enabled.

The adversary Adv acts on behalf of all the dishonest participants. Her symbolic strategy has access to the current (stripped) run, and to a random sequence r_{Adv}. Following the Dolev-Yao model, Adv controls the scheduling among all participants. At each moment, Adv can see all the sequences of moves Λ^s_i chosen by each honest A_i, and can perform *exactly* one action, denoted as λ^s. This can be one of the actions in some Λ^s_i (represented as a pair (A_i, j), where $j \leq |\Lambda^s_i|$), or one action α not requiring the authorization of an honest participant, or a delay δ if all the honest participants agree.

Definition 11 (Symbolic adversary strategies). A symbolic adversary strategy is a PPTIME algorithm $\Sigma^s_{\mathsf{Adv}}(R^s_*, r_{\mathsf{Adv}}, \boldsymbol{\Lambda^s})$, taking as input a stripped symbolic run R^s_*, a random sequence r_{Adv}, and a list $\boldsymbol{\Lambda^s} = \Lambda^s_1 \cdots \Lambda^s_k$ of sequences of stripped moves for each $\mathsf{A}_i \in \mathsf{Hon} = \{\mathsf{A}_1, \ldots, \mathsf{A}_k\}$. The output is a single adversary action λ^s such that, for all symbolic runs R^s, if $\Sigma^s_{\mathsf{Adv}}(\mathit{strip}(R^s), r_{\mathsf{Adv}}, \boldsymbol{\Lambda^s}) = \lambda^s$, one of the following holds:

1) $\lambda^s = (\mathsf{A}_i, j)$, $\Lambda^s_i = \alpha_1 \cdots \alpha_m$, and $\alpha_j = \mathsf{A}_i : \cdots$;
2) $\lambda^s = \alpha$, $\alpha \neq \mathsf{A} : \cdots$ for any A, $\alpha \notin \mathbb{N}$, and $R^s \xrightarrow{\alpha}$;
3) $\lambda^s = \mathsf{B} : \cdots$, $\lambda^s \neq \mathsf{B} : a$ for any a, $\mathsf{B} \notin \mathsf{Hon}$ and $R^s \xrightarrow{\lambda^s}$;
4) $\lambda^s = \delta$, where $\forall i \in 1..k : \big(\Lambda^s_i = \emptyset$ or $\exists \delta_i \in \Lambda^s_i : \delta_i \geq \delta\big)$.
5) $\lambda^s = \mathsf{B} : a$, where (i) $\mathsf{B} \notin \mathsf{Hon}$; (ii) $\Gamma_{\mathit{strip}(R^s)}$ contains $\{\mathsf{B} : a\#\bot\}$; (iii) for some prefix R^s_* of $\mathit{strip}(R^s)$, $\Sigma^s_{\mathsf{Adv}}(R^s_*, r_{\mathsf{Adv}}, \epsilon) = \mathsf{B} : \{G\}C, \Delta$ where $\{\mathsf{B} : a\#N\}$ in Δ, for some $N \neq \bot$ and $\{G\}C$;

If $\Sigma^s_{\mathsf{Adv}}(\mathit{strip}(R^s), r_{\mathsf{Adv}}, \boldsymbol{\Lambda^s}) = \mathsf{B} : \{G\}C, \Delta$ for $\mathsf{B} \notin \mathsf{Hon}$, we ask $\Sigma^s_{\mathsf{Adv}}(\mathit{strip}(R^s), r_{\mathsf{Adv}}, \epsilon) = \mathsf{B} : \{G\}C, \Delta$. ◇

Conditions (1)–(4) are straightforward; in (5) the adversary makes a dishonest participant B reveal a secret a. In such case, we require that the (stripped) run contains the corresponding committed secret $\{\mathsf{B} : a\#\bot\}$, and that, at some previous point R^s_* in the (stripped) run, Σ^s_{Adv} chose a non-$\bot$ length N for a. This requirement is achieved by considering $\Sigma^s_{\mathsf{Adv}}(R^s_*, r_{\mathsf{Adv}}, \epsilon)$, where the use of ϵ is due to the ignorance of the parameter $\boldsymbol{\Lambda^s}$ generated at the point R^s_*. We consequently require that the N chosen by Σ^s_{Adv} does not depend on $\boldsymbol{\Lambda^s}$. This restriction does *not* limit the power of Adv, who can first perform a sequence of actions λ^s depending on $\boldsymbol{\Lambda^s}$, appending them to R^s, and then choose N depending on such actions λ^s.

We now characterise the runs obtained under an adversary with strategy Σ^s_{Adv} taking as input the outputs of the participant strategies. This notion, called *conformance*, also involves a randomness source, fed as input to the strategies.

Definition 12 (Symbolic conformance). Let $\boldsymbol{\Sigma}^s$ be a set of symbolic strategies, comprising those of honest participants $\mathsf{A}_1, \ldots, \mathsf{A}_k$ and of the adversary, and let r be a randomness source. We say that a symbolic run R^s *conforms to* $(\boldsymbol{\Sigma}^s, r)$ when one of the following conditions hold:

1) $R^s = (\Gamma, 0)$, with Γ initial;
2) $\dot{R}^s \xrightarrow{\alpha} R^s$, where $\dot{R}^s$ conforms to $(\boldsymbol{\Sigma}^s, r)$, and, given $\Lambda^s_i = \Sigma^s_{\mathsf{A}_i}(\mathit{strip}(\dot{R}^s), r_{\mathsf{A}_i})$ for all $i \in 1..k$ and $\tilde{\boldsymbol{\Lambda}}^s = \mathit{strip}(\Lambda^s_1 \cdots \Lambda^s_k)$, if $\lambda^s = \Sigma^s_{\mathsf{Adv}}(\mathit{strip}(\dot{R}^s), r_{\mathsf{Adv}}, \tilde{\boldsymbol{\Lambda}}^s)$ then $\lambda^s = \alpha$, or $\lambda^s = (\mathsf{A}_i, j)$ for $\alpha = \alpha_j$ and $\Lambda^s_i = \alpha_1 \cdots \alpha_m$.

If $\boldsymbol{\Sigma}^s$ does not comprise the adversary strategy, we say that R^s conforms to $(\boldsymbol{\Sigma}^s, r)$ when there exists some strategy Σ^s_{Adv} such that R^s conforms to $(\boldsymbol{\Sigma}^s \Sigma^s_{\mathsf{Adv}}, r)$. ◇

IV. Computational model

In this section we introduce our computational model, which will be the target of the BitML compiler. We rely on [26] for a formal model of Bitcoin transactions and its blockchain. Here we just briefly recap these notions, providing some intuition.

In Bitcoin, *transactions* describe transfers of currency (Ƀ). The log of all transactions is maintained on a public, immutable and decentralised data structure called *blockchain*. We represent transactions as records, with fields in, wit, out and absLock. Consider e.g. the transactions T_1 and T_2 below:

T_1	
in:	$\cdots$
wit:	$\cdots$
out:	$0 \mapsto (\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(\mathsf{A})}(\varsigma), v_0\text{Ƀ})$ $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{\mathbf{K}(\mathsf{B})}(\varsigma), v_1\text{Ƀ})$

T_2	
in:	$0 \mapsto (\mathsf{T}_1, 0)$
wit:	$0 \mapsto \mathit{sig}_{\mathbf{K}(\mathsf{A})}$
out:	$0 \mapsto (\lambda x.H(x) = k, v_0\text{Ƀ})$
absLock:	t

The transaction T_1 has two *outputs*: the v_0Ƀ in $\mathsf{out}(0)$ can be redeemed by any transaction T whose in field refers to $(\mathsf{T}_1, 0)$, and whose wit field satisfies the predicate in $\mathsf{out}(0)$ (similarly for the other output). This is the case e.g. of the transaction T_2 above. Its *witness* $\mathit{sig}_{\mathbf{K}(\mathsf{A})}$ is the signature of A on T_2 (excluding the wit field itself), as required by $\mathsf{T}_1.\mathsf{out}(0)$.

If T_1 is on the blockchain and its $\mathsf{out}(0)$ is unspent, A can update the blockchain by appending T_2. This moves v_0Ƀ[1] from T_1 to T_2. The transaction T_2 has only one output, which can be redeemed by any transaction providing a witness having hash k. The time t in $\mathsf{T}_2.\mathsf{absLock}$ represents the earliest moment when T_2 can be put on the blockchain. A subsequent transaction can redeem $(v_1 + v_0)$Ƀ in a single shot. This requires two inputs, $(\mathsf{T}_1, 1)$ and $(\mathsf{T}_2, 0)$, and two witnesses. The witness associated to the first input is a signature of B; the other is a preimage of k.

Definition 13 (Blockchain). A blockchain $\mathbf{B}$ is a sequence $(\mathsf{T}_0, t_0) \cdots (\mathsf{T}_n, t_n)$, where T_0 is *coinbase* transaction (i.e., $\mathsf{T}_0.\mathsf{in} = \bot$), and $t_i \leq t_j$ for all $0 \leq i \leq j \leq n$. ◇

Hereafter, we assume that blockchains are *consistent*, i.e. obtained by appending transactions respecting the Bitcoin protocol (as formalised in [26]). Borrowing from [26], we write $\mathbf{B} \rhd (\mathsf{T}, t)$ when (T, t) consistently updates $\mathbf{B}$.

We now introduce the computational counterparts of symbolic runs and strategies. Our model uses PPTIME algorithms w.r.t. a security parameter $\eta \in \mathbb{N}$. We follow the random oracle model [27] for cryptographic hashes. Namely, a special entity $\mathsf{O} \notin \mathsf{Part}$ provides to every A (including Adv) the access to a hash functionality. To compute a hash, A sends a message $\mathsf{A} \to \mathsf{O} : m$, and waits for the reply $\mathsf{O} \to \mathsf{A} : H(m)$, where $H(m)$ comprises η random bits. We extend the randomness

[1] In the actual Bitcoin, the value the outputs of T_2 must be strictly smaller than v_0, and the difference is paid to the Bitcoin network. For simplicity, in this paper (as in [26]) we neglect these *transaction fees*.

source also to O, assuming $r_{\mathsf{O}} \in \{0,1\}^\omega$ to be defined. For simplicity, we assume that all communications (except those involving O) are (reliable) broadcasts.

Definition 14 (Computational labels). A computational label λ^c is a bitstring, encoding one of the following actions, where $\mathsf{A} \in \mathsf{Part} \cup \{\mathsf{Adv}\}$, and m is a bitstring:

$\mathsf{A} \to * : m$	A broadcasts message m
$\mathsf{A} \to \mathsf{O} : m$	A queries the oracle with message m
$\mathsf{O} \to \mathsf{A} : m$	O answers to A with message m
T	put on the ledger a Bitcoin transaction
δ	perform a delay

◇

We associate each A with two key pairs: K_{A} for signing messages, and $\hat{K}_{\mathsf{A}}$ for redeeming deposits. These key pairs are generated through r_{A}, if $\mathsf{A} \in \mathsf{Hon}$. We write $K_{\mathsf{A}}(r_{\mathsf{A}})$, $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$ for the key pairs generated using the first 2η bits of r_{A}. For the participants *not* in Hon, denoted with $\mathsf{B}_1, \ldots, \mathsf{B}_k$, we write $K_{\mathsf{B}_i}(r_{\mathsf{Adv}})$, $\hat{K}_{\mathsf{B}_i}(r_{\mathsf{Adv}})$ for the key pairs generated from r_{Adv} using the i-th 2η bits. Given any key pair K, we denote with K^p its public part, and with K^s its private part.

Definition 15 (Computational runs). Let r be a randomness source. A computational run R^c is a finite sequence of computational labels, beginning with a prefix R^c_0 such as:

$$\mathsf{T}_0 \cdots \mathsf{A}_i \to * : (K^p_{\mathsf{A}_i}(r_{\mathsf{A}_i}), \hat{K}^p_{\mathsf{A}_i}(r_{\mathsf{A}_i})) \cdots \quad (\forall \mathsf{A}_i \in \mathsf{Hon})$$
$$\cdots \mathsf{B}_j \to * : (K^p_{\mathsf{B}_j}(r_{\mathsf{Adv}}), \hat{K}^p_{\mathsf{B}_j}(r_{\mathsf{Adv}})) \cdots \quad (\forall \mathsf{B}_j \notin \mathsf{Hon})$$

where T_0 is a coinbase transaction, and for each participant P (honest or dishonest), there exists an output of T_0 redeemable with the private key $\hat{K}^s_{\mathsf{P}}$. We say that the run R^c_0 is *initial*. We denote with δ_{R^c} the sum of all the delays in R^c. ◇

The labels $\mathsf{P}_i \to * : (K^p_{\mathsf{P}_i}, \hat{K}^p_{\mathsf{P}_i})$ represent a broadcast of P_i's public keys (of both kinds) to all participants. In this way, each participant starts with some funds (possibly 0), and knows the public keys of the others.

Analogously to Definition 8, we define the stripping of a computational run. We denote with $strip_{\mathsf{A}}(R^c)$ the run obtained by removing from R^c all the messages *not* visible by A, i.e. the messages between O and some other $\mathsf{B} \neq \mathsf{A}$.

Definition 16 (Stripping of computational runs). For each computational run R^c and participant $\mathsf{A} \in \mathsf{Part} \cup \{\mathsf{Adv}\}$, we define $strip_{\mathsf{A}}(R^c)$ as the computational run obtained from R^c by replacing each label λ^c with $strip_{\mathsf{A}}(\lambda^c)$, defined as follows:

$$strip_{\mathsf{A}}(\lambda^c) = \begin{cases} \epsilon & \text{if } \lambda^c = \mathsf{B} \to \mathsf{O} : m, \text{ with } \mathsf{B} \neq \mathsf{A} \\ \epsilon & \text{if } \lambda^c = \mathsf{O} \to \mathsf{B} : m, \text{ with } \mathsf{B} \neq \mathsf{A} \\ \lambda^c & \text{otherwise} \end{cases}$$ ◇

By extracting the transactions from a computational run, we obtain a blockchain, whose consistency will be ensured by Definition 20. This will be used to define strategies.

Definition 17 (Blockchain of a computational run). For each R^c, we define the blockchain $\mathbf{B}_{R^c}$ inductively as follows:

$$\mathbf{B}_{\mathsf{T}_0} = (\mathsf{T}_0, 0) \qquad \mathbf{B}_{R^c \lambda^c} = \begin{cases} \mathbf{B}_{R^c}(\mathsf{T}, \delta_{R^c}) & \text{if } \lambda^c = \mathsf{T} \\ \mathbf{B}_{R^c} & \text{otherwise} \end{cases}$$ ◇

Definition 18 (Computational participant strategies). The computational strategy of a participant $\mathsf{A} \in \mathsf{Hon}$ is a PPTIME algorithm $\Sigma^c_{\mathsf{A}}(R^c_*, r_{\mathsf{A}})$, taking as input a ($\mathsf{A}$-stripped) computational run R^c_* and a random sequence r_{A}. The output is a *finite* set Λ^c of computational labels, such that if $\lambda^c \in \Lambda^c$, then one of the following items holds:

1) $\lambda^c = \mathsf{A} \to * : m$ or $\lambda^c = \mathsf{A} \to \mathsf{O} : m$, for some m;
2) $\lambda^c = \mathsf{T}$, where $\mathbf{B}_{R^c_*} \rhd (\mathsf{T}, \delta_{R^c_*})$, where in R^c_* we can find (in this order): (i) all the inputs of T; (ii) a message $\mathsf{B} \to * : \mathsf{T}$, for some B; (iii) for each witness w in T, a message $\mathsf{B} \to * : w$, for some B.
3) $\lambda^c = \delta$.

We further require that participant strategies are persistent: i.e., if $\Lambda^c = \Sigma^c_{\mathsf{A}}(R^c_*, r_{\mathsf{A}})$, then for all λ^c such that $\mathbf{B}_{R^c_* \lambda^c}$ is consistent, $\Sigma^c_{\mathsf{A}}(R^c_* \lambda^c, r_{\mathsf{A}})$ includes the computational labels:

$$\{\mathsf{T} \in \Lambda^c \mid \mathbf{B}_{R^c_* \lambda^c} \rhd (\mathsf{T}, \delta_{R^c_* \lambda^c})\} \cup$$
$$\{\mathsf{A} \to * : m \in \Lambda^c \mid (\mathsf{A} \to * : m) \neq \lambda^c\} \cup$$
$$\{\mathsf{A} \to \mathsf{O} : m \in \Lambda^c \mid (\mathsf{A} \to \mathsf{O} : m) \neq \lambda^c\}$$ ◇

Persistency ensures that, if A at a certain point wants to perform some λ^c (sending m or putting T on the blockchain), she cannot change her mind in the future, until λ^c is performed. Condition (2) requires that, before being able to append T, one has to make both T and its witnesses public. This models the fact that, before T actually appears on the blockchain, it has to be broadcast on the Bitcoin network, potentially enabling an adversary to know T beforehand.

Definition 19 (Computational adversary strategies). A computational adversary strategy is a PPTIME algorithm $\Sigma^c_{\mathsf{Adv}}(R^c_*, r_{\mathsf{Adv}}, \mathbf{\Lambda^c})$, taking as input a ($\mathsf{Adv}$-stripped) computational run R^c_*, a random sequence r_{Adv}, and a list $\mathbf{\Lambda^c}$ of sequences of moves of each $\mathsf{A}_i \in \mathsf{Hon} = \{\mathsf{A}_1, \ldots, \mathsf{A}_k\}$. The output is a single computational label λ^c such that if $\Sigma^c_{\mathsf{Adv}}(R^c_*, r_{\mathsf{Adv}}, \mathbf{\Lambda^c}) = \lambda^c$, one of the following holds:

1) $\lambda^c = \mathsf{A} \to * : m$ or $\lambda^c = \mathsf{Adv} \to \mathsf{O} : m$, for some m, A;
2) $\lambda^c = \mathsf{T}$, where $\mathbf{B}_{R^c_*} \rhd (\mathsf{T}, \delta_{R^c_*})$, where in R^c_* we can find (in this order): (i) all the inputs of T; (ii) a message $\mathsf{B} \to * : \mathsf{T}$, for some B; (iii) for each witness w in T, a message $\mathsf{B} \to * : w$, for some B.
3) $\lambda^c = \delta$, where $\forall i \leq k : \big(\Lambda^c_i = \emptyset$ or $\exists \delta_i \in \Lambda^c_i : \delta_i \geq \delta\big)$. ◇

Note that (1) allows Adv to impersonate any A, to use either A's messages in Λ^c, or self-produced ones. The other cases allow Adv to (consistently) extend the blockchain, and to delay. (if all the honest participants agree).

Given a set of strategies $\mathbf{\Sigma}^c$ for all the honest participants and for the adversary, and a randomness source r, we now define which runs R^c can result from making everyone interact. Such runs are said to *conform* to $(\mathbf{\Sigma}^c, r)$.

Definition 20 (Computational conformance). Let $\mathbf{\Sigma}^c$ be a set of computational strategies, including those of the honest participants $\mathsf{A}_1, \ldots, \mathsf{A}_k$ and of the adversary, and let r be a randomness source. We say that a computational run R^c *pre-conforms to* $(\mathbf{\Sigma}^c, r)$ if one of the following conditions holds:

1) R^c is initial;
2) $R^c = \dot{R}^c \lambda^c$, where $\dot{R}^c$ pre-conforms to $(\mathbf{\Sigma}^c, r)$, and $\lambda^c = \Sigma^c_{\mathsf{Adv}}(strip_{\mathsf{Adv}}(\dot{R}^c), r_{\mathsf{Adv}}, \mathbf{\Lambda^c})$, where $\mathbf{\Lambda^c} = \Lambda^c_1 \cdots \Lambda^c_k$, $\Lambda^c_i = \Sigma^c_{\mathsf{A}_i}(strip_{\mathsf{A}_i}(\dot{R}^c), r_{\mathsf{A}_i})$, O not occurring in $\mathbf{\Lambda^c}$, λ^c.
3) $R^c = \dot{R}^c(\mathsf{Adv} \to \mathsf{O} : m)(\mathsf{O} \to \mathsf{Adv} : h_m)$, where $\dot{R}^c$ pre-conforms to $(\mathbf{\Sigma}^c, r)$, and $\Sigma^c_{\mathsf{Adv}}(strip_{\mathsf{Adv}}(\dot{R}^c), r_{\mathsf{Adv}}, \mathbf{\Lambda^c}) = \mathsf{Adv} \to \mathsf{O} : m$, where $\mathbf{\Lambda^c} = \Lambda^c_1 \cdots \Lambda^c_k$, $\Lambda^c_i = \Sigma^c_{\mathsf{A}_i}(strip_{\mathsf{A}_i}(\dot{R}^c), r_{\mathsf{A}_i})$, and O does not occur in $\mathbf{\Lambda^c}$.
4) $R^c = \dot{R}^c(\mathsf{A}_j \to \mathsf{O} : m)(\mathsf{O} \to \mathsf{A}_j : h_m)$, where $\dot{R}^c$ pre-conforms to $(\mathbf{\Sigma}^c, r)$, $\Lambda^c_i = \Sigma^c_{\mathsf{A}_i}(strip_{\mathsf{A}_i}(\dot{R}^c), r_{\mathsf{A}_i})$, and $(\mathsf{A}_j \to \mathsf{O} : m)$ is the first occurrence of a query to the oracle in $\Lambda^c_1 \cdots \Lambda^c_k$.

Further, in both Items 3 and 4, given n the number of distinct queries to O in R^c, we require that if m was already requested, then h_m is its reply in $\dot{R}^c$; otherwise, h_m is the portion of r_{O} of length η starting from $n\eta$.

We say that R^c *conforms to* $(\mathbf{\Sigma}^c, r)$ if R^c is a prefix of a run pre-conforming to $(\mathbf{\Sigma}^c, r)$. ◇

Above, in Items 3 and 4 we handle the queries to O, modelling the hash functionality as in the random oracle model. We let queries have higher priority than other actions.

V. Compiling contracts to Bitcoin transactions

We compile a contract advertisement $\{G\}C$ into a sequence of standard Bitcoin transactions[2]. Our compiler relies on the following parameters, which depend on G and C:

- PartG is the set of all participants occurring in G;
- $\mathbf{part}$ maps deposit names in G to the corresponding participants (e.g., $\mathbf{part}(x) = \mathsf{A}$ if $\mathsf{A} : ? \, v \, @ \, x$ in G);
- $\mathbf{txout}$ maps deposit names in G to the corresponding Bitcoin transaction output (T, o);
- $\mathbf{val}$ maps deposit names in G to the value contained in the deposit (e.g., $\mathbf{val}(x) = v$ if $\mathsf{A} : ? \, v \, @ \, x$ in G);
- $\mathbf{sechash}$ maps secret names in G to the corresponding committed hashes.

Further, we assume that participants generate the following key pairs, and exchange their public parts:

- $\mathbf{K}(\mathsf{A})$, for each $\mathsf{A} \in \mathsf{PartG}$;
- $\mathbf{K}(D, \mathsf{A})$, for each subterm D of C, and each $\mathsf{A} \in \mathsf{PartG}$.

For a set of participants $\mathcal{P} = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$, we denote with $\mathbf{K}(D, \mathcal{P})$ the set of key pairs $\{\mathbf{K}(D, \mathsf{A}_1), \ldots, \mathbf{K}(D, \mathsf{A}_n)\}$.

Definition 21 (BitML compiler). The function $\mathbb{B}_{\mathbf{adv}}(\cdot)$ from contract advertisements to sequences of Bitcoin transactions is defined by the rules in Figure 6. ◇

[2]Standard Bitcoin transactions are obtained from those obtained by our compiler using the tool https://github.com/bitcoin-transaction-model. This is crucial, since the Bitcoin network currently discards non-standard transactions.

The function $\mathbb{B}_{\mathbf{adv}}$ produces all the transactions for $\{G\}C$. In particular, T_{init} will be the first to be put on the blockchain, representing the stipulation of C. Function $\mathbb{B}_{\mathbf{C}}$ assigns to a contract $C = \sum_i D_i$ a transaction T_C, which can be redeemed only by using the keys of the subterms D_i of C. The functions $\mathbb{B}_{\mathbf{D}}$ and $\mathbb{B}_{\mathbf{out}}$ handle the possible actions of each D. The action `split` is handled by $\mathbb{B}_{\mathbf{par}}$. We provide more intuition through an example.

Example 7. We compile the timed commitment $\{G\}C$ of Example 3, where $G = \mathsf{A} : ! \, v \, @ \, x \mid \mathsf{A} : \mathtt{secret} \, a \mid \mathsf{B} : ! \, 0 \, @ \, y$, and $C = D_1 + D_2$, with $D_1 = \mathtt{reveal} \, a . \, \mathtt{withdraw} \, \mathsf{A}$ and $D_2 = \mathtt{after} \, t : \mathtt{withdraw} \, \mathsf{B}$. Assume that: $\mathsf{A} \in \mathsf{Hon}$, $\mathbf{txout}(x) = (\mathsf{T}_{\mathsf{A}}, 0)$ for some T_{A} whose output 0 has value v and is redeemable by A, $\mathbf{sechash}(a) = h_a$, $\mathbf{val}(x) = v$. Similarly, for B: $\mathbf{txout}(y) = (\mathsf{T}_{\mathsf{B}}, 0)$, and $\mathbf{val}(y) = 0$.

The compiler produces $\mathbb{B}_{\mathbf{adv}}(\{G\}C) = \mathsf{T}_{init}\mathcal{T}_1\mathcal{T}_2$, where:

$$\begin{aligned}
\mathcal{T}_1 &= \mathbb{B}_{\mathbf{D}}(D_1, D_1, \mathsf{T}_{init}, 0, v, \{\mathsf{A}, \mathsf{B}\}, 0) \\
&= \mathbb{B}_{\mathbf{C}}(\mathtt{withdraw} \, \mathsf{A}, D_1, \mathsf{T}_{init}, 0, v, \emptyset, \{\mathsf{A}, \mathsf{B}\}, 0) \\
&= \mathsf{T}'\mathbb{B}_{\mathbf{D}}(\mathtt{withdraw} \, \mathsf{A}, \mathtt{withdraw} \, \mathsf{A}, \mathsf{T}', 0, v, \{\mathsf{A}, \mathsf{B}\}, 0) = \mathsf{T}'\mathsf{T}'_{\mathsf{A}} \\
\mathcal{T}_2 &= \mathbb{B}_{\mathbf{D}}(D_2, D_2, \mathsf{T}_{init}, 0, v, \{\mathsf{A}, \mathsf{B}\}, 0) \\
&= \mathbb{B}_{\mathbf{D}}(\mathtt{withdraw} \, \mathsf{B}, D_2, \mathsf{T}_{init}, 0, v, \{\mathsf{A}, \mathsf{B}\}, t) = \mathsf{T}'_{\mathsf{B}}
\end{aligned}$$

The obtained transactions are in Figure 7, where:

$$\begin{aligned}
e_1 &= \mathbb{B}_{\mathbf{out}}(D_1) \\
&= \mathsf{versig}_{\mathbf{K}(D_1, \{\mathsf{A},\mathsf{B}\})}(\varsigma_{\mathsf{A}}\varsigma_{\mathsf{B}}) \wedge \mathsf{H}(b) = h_a \wedge |b| \geq \eta \\
e_2 &= \mathbb{B}_{\mathbf{out}}(D_2) = \mathsf{versig}_{\mathbf{K}(D_2, \{\mathsf{A},\mathsf{B}\})}(\varsigma_{\mathsf{A}}\varsigma_{\mathsf{B}}) \\
e' &= \mathbb{B}_{\mathbf{out}}(\mathtt{withdraw} \, \mathsf{A}) = \mathsf{versig}_{\mathbf{K}(\mathtt{withdraw} \, \mathsf{A}, \{\mathsf{A},\mathsf{B}\})}(\varsigma_{\mathsf{A}}\varsigma_{\mathsf{B}})
\end{aligned}$$

For the sake of readability we do not use distinct variables for different signatures of the same participant. The participants start by generating the transactions off chain, and exchanging the signatures shown in Figure 7. Doing this, A commits to her secret, whose hash h_a occurs in the output script T_{init}.out. After that, both A and B sign T_{init}, and put it on the ledger, stipulating the contract. The transaction T_{init} can be redeemed either by T' or by T'_{B}. In the first case, T' has to add to her witness a value a such that $H(a) = h_a$ and $|a| \geq \eta$. After that, A can redeem her deposit (now in T') by putting T'_{A} on the blockchain. In the second case, T_{init} can be redeemed by T'_{B}: however, this transaction can be put on the blockchain only after t, because of the timelock in T'_{B}. ◇

VI. Security of the compiler

In this section we prove the security of our compiler. We start with two auxiliary definitions: coherence between symbolic and computational runs (Section VI-A), and a mapping from symbolic to computational strategies (Section VI-B). Our computational soundess result (Theorem 1) is in Section VI-C.

A. Relating symbolic and computational runs

Two runs R^s and R^c are coherent when each symbolic step in R^s is matched by its computational implementation in R^c. We provide some intuition by discussing a few cases. Advertising a contract (rule [C-Advertise]) corresponds to a broadcast of a suitable bitstring, where symbolic deposit names

$$\frac{\begin{array}{l} G = \left(\|_{i\in I} \mathsf{A}_i : ?\, v_i \,@\, x_i\right) \mid \left(\|_{i\in J} \mathsf{B}_i : !\, v'_i \,@\, y_i\right) \mid \left(\|_{i\in K} \mathsf{C}_i : \mathtt{secret}\, a_i\right) \\ C = \sum_{i=1}^{m} D_i \qquad\qquad v = \sum_{i\in J} v'_i \\ e_i = \ddot{\mathtt{B}}_{\mathbf{out}}(D_i) \;\; (\forall i \in 1..m) \qquad \boldsymbol{x} = \biguplus_{i=1}^{m} \mathrm{fv}(e_i) \\ \mathcal{T}_i = \ddot{\mathtt{B}}_{\mathbf{D}}(D_i, D_i, \mathsf{T}_{init}, 0, v, \mathsf{PartG}, 0) \;\; (\forall i \in 1..m) \end{array}}{\ddot{\mathtt{B}}_{\mathbf{adv}}(\{G\}C) = \mathsf{T}_{init}\,\mathcal{T}_1 \cdots \mathcal{T}_m}$$

T_{init}
in: $i \mapsto \mathbf{txout}(y_i) \;\; (\forall i \in J)$
wit: $\bot$
out: $(\lambda \boldsymbol{x}.\, \bigvee_{i=1}^{m} e_i,\, v)$

$$\frac{\begin{array}{l} C = \sum_{i=1}^{m} D_i \qquad I = \{z_1, \ldots, z_k\} \\ e_i = \ddot{\mathtt{B}}_{\mathbf{out}}(D_i) \;\; (\forall i \in 1..m) \\ \boldsymbol{x} = \biguplus_{i=1}^{m} \mathrm{fv}(e_i) \\ \mathcal{T}_i = \ddot{\mathtt{B}}_{\mathbf{D}}(D_i, D_i, \mathsf{T}_C, 0, v, \mathsf{PartG}, 0) \;\; (\forall i \in 1..m) \end{array}}{\ddot{\mathtt{B}}_{\mathbf{C}}(C, D_p, \mathsf{T}, o, v, I, \mathcal{P}, t) = \mathsf{T}_C\,\mathcal{T}_1 \cdots \mathcal{T}_m}$$

T_C
in: $0 \mapsto (\mathsf{T}, o),\; i \mapsto \mathbf{txout}(z_i) \;\; (\forall i \in 1..k)$
wit: $0 \mapsto sig_{\mathbf{K}(D_p,\mathcal{P})},\; i \mapsto sig_{\mathbf{K}(\mathbf{part}(z_i))} \;\; (\forall i \in 1..k)$
out: $(\lambda \boldsymbol{x}.\, \bigvee_{i=1}^{m} e_i,\, v)$
absLock: t

$$\frac{D \not\equiv \mathsf{A}_1 : \cdots : \mathsf{A}_n : \mathtt{after}\, t_1 : \cdots : \mathtt{after}\, t_k : \mathtt{put}\, \boldsymbol{z}\, \&\, \mathtt{reveal}\, \boldsymbol{a}\, \mathtt{if}\, p.\, C \qquad \boldsymbol{\varsigma} \text{ fresh}}{\ddot{\mathtt{B}}_{\mathbf{out}}(D) = \mathsf{versig}_{\mathbf{K}(D,\mathsf{PartG})}(\boldsymbol{\varsigma})}$$

$$\frac{D \equiv \mathsf{A}_1 : \cdots : \mathsf{A}_n : \mathtt{after}\, t_1 : \cdots : \mathtt{after}\, t_k : \mathtt{put}\, \boldsymbol{z}\, \&\, \mathtt{reveal}\, \boldsymbol{a}\, \mathtt{if}\, p.\, C \qquad \boldsymbol{a} = a_1 \cdots a_m \qquad \boldsymbol{\varsigma}, b_1 \cdots b_m \text{ fresh}}{\ddot{\mathtt{B}}_{\mathbf{out}}(D) = \mathsf{versig}_{\mathbf{K}(D,\mathsf{PartG})}(\boldsymbol{\varsigma}) \;\wedge\; \ddot{\mathtt{B}}(p) \;\wedge\; \bigwedge_{i=1}^{m} \mathsf{H}(b_i) = \mathbf{sechash}(a_i) \;\wedge\; |b_i| \geq \eta}$$

$$\frac{D = \mathtt{withdraw}\, \mathsf{A}}{\ddot{\mathtt{B}}_{\mathbf{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \{\mathsf{in} : (\mathsf{T}, o),\; \mathsf{wit} : sig_{\mathbf{K}(D_p,\mathcal{P})},\; \mathsf{out} : (\lambda\varsigma.\, \mathsf{versig}_{\mathbf{K}(\mathsf{A})}(\varsigma),\, v),\; \mathsf{absLock} : t\}}$$

$$\frac{D = \mathtt{put}\, \boldsymbol{z}\, \&\, \mathtt{reveal}\, \boldsymbol{a}\, \mathtt{if}\, p.\, C \qquad v' = v + \sum_{x\in \boldsymbol{z}} \mathbf{val}(x)}{\ddot{\mathtt{B}}_{\mathbf{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \ddot{\mathtt{B}}_{\mathbf{C}}(C, D_p, \mathsf{T}, o, v', \boldsymbol{z}, \mathcal{P}, t)} \qquad \frac{D = \mathtt{split}\, \boldsymbol{v} \to \boldsymbol{C} \qquad \boldsymbol{v} = v_1 \cdots v_k \qquad \sum_{i=1}^{k} v_i \leq v}{\ddot{\mathtt{B}}_{\mathbf{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \ddot{\mathtt{B}}_{\mathbf{par}}(\boldsymbol{C}, D_p, \mathsf{T}, o, \boldsymbol{v}, \mathcal{P}, t)}$$

$$\frac{D = \mathsf{A} : D'}{\ddot{\mathtt{B}}_{\mathbf{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \ddot{\mathtt{B}}_{\mathbf{D}}(D', D_p, \mathsf{T}, o, v, \mathcal{P} \setminus \{\mathsf{A}\}, t)} \qquad \frac{D = \mathtt{after}\, t' : D'}{\ddot{\mathtt{B}}_{\mathbf{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \ddot{\mathtt{B}}_{\mathbf{D}}(D', D_p, \mathsf{T}, o, v, \mathcal{P}, \max\{t, t'\})}$$

$$\frac{\begin{array}{l} \boldsymbol{C} = C_1 \cdots C_n \qquad C_i = \sum_{j=1}^{k_i} D_{i,j} \;\; (\forall i \in 1..n) \\ \boldsymbol{v} = v_1 \cdots v_n \qquad e_{i,j} = \ddot{\mathtt{B}}_{\mathbf{out}}(D_{i,j}) \;\; (\forall i \in 1..n, j \in 1..k_i) \\ \boldsymbol{x}_i = \biguplus_{j=1}^{k_i} \mathrm{fv}(e_{i,j}) \;\; (\forall i \in 1..n) \\ \mathcal{T}_{i,j} = \ddot{\mathtt{B}}_{\mathbf{D}}(D_{i,j}, D_{i,j}, \mathsf{T}_C, i-1, v_i, \mathsf{PartG}, 0) \;\; (\forall i \in 1..n, j \in 1..k_i) \end{array}}{\ddot{\mathtt{B}}_{\mathbf{par}}(\boldsymbol{C}, D_p, \mathsf{T}, o, \boldsymbol{v}, \mathcal{P}, t) = \mathsf{T}_C(\mathcal{T}_{i,j})_{i\in 1..n,\, j\in 1..k_i}}$$

T_C
in: (T, o)
wit: $sig_{\mathbf{K}(D_p,\mathcal{P})}$
out: $i-1 \mapsto (\lambda \boldsymbol{x}_i.\, \bigvee_{j=1}^{k_i} e_{i,j},\, v_i) \;\; (\forall i \in 1..n)$
absLock: t

$$\ddot{\mathtt{B}}(true) = true \qquad \ddot{\mathtt{B}}(p_1 \wedge p_2) = \ddot{\mathtt{B}}(p_1) \wedge \ddot{\mathtt{B}}(p_2) \qquad \ddot{\mathtt{B}}(\neg p) = \neg \ddot{\mathtt{B}}(p) \qquad \ddot{\mathtt{B}}(e_1 \circ e_2) = \ddot{\mathtt{B}}(e_1) \circ \ddot{\mathtt{B}}(e_2)$$

$$\ddot{\mathtt{B}}(N) = N \qquad \ddot{\mathtt{B}}(|a|) = |a| - \eta \qquad \ddot{\mathtt{B}}(e_1 \bullet e_2) = \ddot{\mathtt{B}}(e_1) \bullet \ddot{\mathtt{B}}(e_2)$$

Fig. 6: Compiling contracts to Bitcoin transactions.

T_{init}
in: $0 \mapsto (\mathsf{T}_\mathsf{A}, 0), 1 \mapsto (\mathsf{T}_\mathsf{B}, 0)$
wit: $\bot$
out: $(\lambda\varsigma_\mathsf{A}\varsigma_\mathsf{B}.e_1 \vee e_2,\, v)$

T'
in: $(\mathsf{T}_{init}, 0)$
wit: $sig_{\mathbf{K}(D_1,\{\mathsf{A},\mathsf{B}\})}$
out: $(\lambda\varsigma_\mathsf{A}\varsigma_\mathsf{B}.e',\, v)$

T'_A
in: $(\mathsf{T}', 0)$
wit: $sig_{\mathbf{K}(\mathtt{withdraw}\ \mathsf{A},\{\mathsf{A},\mathsf{B}\})}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathbf{K}(\mathsf{A})}(\varsigma),\, v)$

T'_B
in: $(\mathsf{T}_{init}, 0)$
wit: $sig_{\mathbf{K}(D_2,\{\mathsf{A},\mathsf{B}\})}$
out: $(\lambda\varsigma.\, \mathsf{versig}_{\mathbf{K}(\mathsf{B})}(\varsigma),\, v)$
absLock: t

Fig. 7: Transactions obtained by compiling the timed commitment contract.

have been encoded as Bitcoin transaction outputs. Committing a secret (rule [C-AuthCommit]) corresponds to a sequence of computational actions: generating a nonce, obtaining its hash (querying O), signing and broadcasting it. Contract stipulation (rule [C-Init]) corresponds to putting on the blockchain the transaction T_{init} obtained by the compiler. Moves related to active contracts (e.g., [C-PutRev]) correspond to putting on the blockchain the related transactions, as generated by the compiler. Authorizing a move (rule [C-AuthControl]) corresponds to broadcasting the needed signature. Revealing a secret (rule [C-AuthRev]) corresponds to broadcasting the associated nonce.

As anticipated in Section I, computational moves without a symbolic representation are ignored by the coherence relation. For instance, the computational Adv can broadcast arbitrary bitstrings, with no symbolic meaning. Adv can also put on the ledger any transaction T allowed by Bitcoin, including those not generated by our compiler, which so has no symbolic counterpart. In such case, coherence might still hold, depending on the inputs of T. If no input of T is symbolically represented, coherence holds, but we associate no corresponding move in R^s. Another case when coherence holds is that in which T redeems transactions which implement deposits $\langle \mathsf{A}, v\rangle$: here, we represent T in R^s using [Dep-Destroy]. Instead, coherence does not hold if T redeems some transaction which implements an active contract $\langle C, v\rangle$. Consequently, when coherence holds, the computational behavior of transactions implementing contracts closely matches their symbolic semantics.

Defining coherence is rather long, since it has to deal with all the possible cases of the symbolic semantics, and technically intricate, since computational runs have no intrinsic structure. For these reasons, we defer the actual definition to Appendix D (Definition 25). This definition exploits three maps, $txout$, $sechash$ and κ, the role of which is similar to that of the compiler parameters. Hereafter we write $R^s \sim_r R^c$ when the two runs are coherent w.r.t. the randomness source r.

B. Mapping symbolic to computational strategies

Given a symbolic strategy Σ^s of an honest participant A, we translate it to a computational strategy Σ^c which emulates its behaviour. We first describe how to translate the stipulation of a contract advertisement $\{G\}C$, i.e. by providing a computational counterpart to the rules in Figure 3. In the computational model, this is performed by following a *stipulation protocol* (Definition 22), which will be exploited to construct Σ^c.

Note that, while in the symbolic setting we model contract advertisements as terms $\{G\}C$, containing names x for the deposits, in the computational setting we encode such advertisements as bitstrings, using transaction outputs (T, o) instead of deposit names x. In the stipulation protocol, we assume as given this representation $\mathcal{C}$, which we call *computational* contract advertisement, and the parameters of Σ^c_{A}, i.e. the computational run R^c and the random sequence r_{A}. After decoding $\mathcal{C}$ as a symbolic advertisement $\{G\}C$, A interacts with the other participants to obtain all the parameters required to compile $\{G\}C$. Then, A computes $\mathbb{B}_{\mathbf{adv}}(\{G\}C)$, and finally she puts the generated T_{init} transaction on the ledger.

Definition 22 (Stipulation protocol). Let $\mathsf{A} \in \mathsf{Hon}$, let R^c_* be a (A-stripped) computational run, and let r_{A} be a random sequence. The stipulation protocol for a computational contract advertisement $\mathcal{C}$ is the following.

1) A decodes $\mathcal{C}$, constructing a symbolic contract advertisement $\{G\}C$; in doing this, A chooses distinct symbolic names for all the transaction outputs in $\mathcal{C}$. The mapping $\mathbf{txout}$ is defined according to the used correspondence between names and transaction outputs.
2) A infers from G the parameters $\mathbf{part}$, PartG, and $\mathbf{val}$.
3) A uses r_{A} to obtain the key pairs $K_{\mathsf{A}}(r_{\mathsf{A}})$ and $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. The key $K_{\mathsf{A}}(r_{\mathsf{A}})$ is used by A to sign all the protocol messages. Further, A reads, from the initial prefix of the run R^c_*, the public keys $K^p_{\mathsf{B}}(r_{\mathsf{B}})$ and $\hat{K}^p_{\mathsf{B}}(r_{\mathsf{B}})$ of the all $\mathsf{B} \in \mathsf{PartG}\backslash\{\mathsf{A}\}$. The keys $K^p_{\mathsf{B}}(r_{\mathsf{B}})$ are then used by A to filter out the incoming messages with incorrect signatures.
4) A defines compiler keys: $\mathbf{K}(\mathsf{A})$ is $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$, while the public part of $\mathbf{K}(\mathsf{B})$ is $\hat{K}^p_{\mathsf{B}}(r_{\mathsf{B}})$ for $\mathsf{B} \neq \mathsf{A}$. The keys $\mathbf{K}(D, \mathsf{A})$ are generated by A consuming $\sim \eta$ fresh bits from r_{A}, and their public parts are shared with the others. Dually, A defines the public parts of keys $\mathbf{K}(D, \mathsf{B})$ using the first broadcasts by the other participants. Let $\boldsymbol{k}$ be the sequence of public keys known by A after this step.
5) A generates from r_{A} a secret nonce of the desired length (to be defined by Σ^c_{A}) for every $\mathsf{A}:\mathtt{secret}\ a$ in G. Then, A computes the hashes $\boldsymbol{h} = h_1 \cdots h_k$ of secret nonces (by querying O), and broadcasts all these hashes as a single message $m(\mathcal{C}, \boldsymbol{h}, \boldsymbol{k})$. Dually, A receives the hashes $\boldsymbol{h'}$ from the other participants. When doing this, A defines $\mathbf{sechash}$ using the first (correctly signed) $m(\mathcal{C}, \boldsymbol{h'}, \boldsymbol{k})$ in R^c_* which has no duplicate hashes, and has no hashes already occurring (signed) in R^c_*.
6) A computes $\mathbb{B}_{\mathbf{adv}}(\{G\}C)$, generating a list of transactions. The signatures by A occurring in the witnesses are computed and shared with others, while the signatures of other participants are received and verified. Note that the T_{init} transaction, (whose wit field is left to $\bot$ by the compiler), is not signed, yet.
7) Only at this point, after having verified all the other signatures, A signs the first transaction, adding the signatures for her persistent deposits using $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. Dually, she receives the signatures from the other participants for their own persistent deposits.
8) Finally, A broadcasts the signed T_{init}, and its witnesses, and puts it on the ledger.

In order to define the mapping from symbolic to computational strategies, we first sketch how to parse a (stripped) computational run R^c_* so to obtain a (stripped) symbolic run R^s_*. This requires to find, when possible, a symbolic counterpart to computational actions, mapping, e.g., computational advertisements to symbolic ones, signatures to authorizations, transactions to symbolic actions (involving deposits or contracts), and secrets to reveal actions. This correspondence closely follows Definition 25. Indeed, to perform the parsing,

we can exploit maps similar to $txout$, $sechash$ and κ to keep track of the correspondence at each step of R^s_*.

Using such maps, we can detect when a transaction T in R^c_* has some input with a symbolic counterpart (i.e. in $\mathrm{ran}\, txout$). When that happens, we can map T into its corresponding action in R^s_*. According to Definitions 18 and 19, T has to be preceded by the broadcasts of its witnesses w, which, in turn, must be preceded by the broadcast of T. This allows to parse the signatures in w, so to generate authorizations in R^s_*.

A few cases must be handled with more care. For instance, Adv could broadcast a signature *before* the signed transaction. When this happens, we simply ignore the message; duplicate signatures are ignored as well. Further, a computational contract advertisement could involve only dishonest participants: we ignore that as well. Adv can consume her own deposits (among those tracked by $txout$), to create an arbitrary transaction without a symbolic counterpart. In this case, in the semantics we use the [DEP-DESTROY] move in R^s_*, preceded by its authorizations, making those deposits disappear from the symbolic world. When the hash of a secret is committed, it can not be parsed precisely as a [C-AUTHCOMMIT] move, since the latter involves the length of the secrets, which can not be inferred from the hash. However, this is not needed, since the *stripped* run R^s_* does not involve such lengths.

Definition 23 (From symbolic to computational strategies). Let Σ^s_{A} be a symbolic strategy, with $\mathsf{A} \in \mathsf{Hon}$. We define $\aleph(\Sigma^s_{\mathsf{A}}) = \Sigma^c_{\mathsf{A}}$ below. Given the parameters R^c_*, r_{A} of Σ^c_{A}, we:

1) parse the (stripped) run R^c_*, so to obtain a corresponding symbolic (stripped) run R^s_*, as sketched above;
2) halve the random sequence r_{A} as $(\pi_1(r_{\mathsf{A}}), \pi_2(r_{\mathsf{A}}))$;
3) evaluate $\Lambda^s = \Sigma^s_{\mathsf{A}}(R^s_*, \pi_1(r_{\mathsf{A}}))$;
4) convert the symbolic actions Λ^s into computational actions Λ^c, and define $\Sigma^c_{\mathsf{A}}(R^c_*, r_{\mathsf{A}}) = \Lambda^c$. When Λ^s contains $\mathsf{A} : \{G\}C, \Delta$ or $\mathsf{A} : \{G\}C, x$, their conversion follows the stipulation protocol (Definition 22), using $\pi_2(r_{\mathsf{A}})$. There, at item 5, we choose the length of each secret by adding η to the corresponding value N in Δ. $\diamond$

C. Computational soundness

Our main result follows. We assume that honest participants have an associated symbolic strategy, and that their computational strategy is consequently obtained through the mapping $\aleph$. We also assume that Bitcoin uses secure cryptographic primitives, namely ideal hash functions (according to the random oracle model), and a digital signature scheme which is robust against *existential forgery* attacks. We also assume that the Bitcoin blockchain is append-only, i.e. that there are no forks, and *consistent*, i.e. miners only append valid transactions [26]. This is a rough abstraction of the security properties established in [28]–[30]

Our computational soundness result establishes that any computational run conforming to the (computational) strategies, with overwhelming probability has a corresponding symbolic run conforming to the (symbolic) strategies.

Theorem 1 (Computational soundness). *Let $\mathbf{\Sigma}^s$ be a set of symbolic strategies for all $\mathsf{A} \in \mathsf{Hon}$. Let $\mathbf{\Sigma}^c$ be a set of computational strategies such that $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$ for all $\mathsf{A} \in \mathsf{Hon}$, and with an arbitrary adversary strategy Σ^c_{Adv}. Fix $k \in \mathbb{N}$. Then, the following set has overwhelming probability:*

$$\left\{ r \;\middle|\; \begin{array}{l} \forall R^c \text{ conforming to } (\mathbf{\Sigma}^c, r) \text{ with } |R^c| \leq \eta^k : \\ \quad \exists R^s \text{ conforming to } (\mathbf{\Sigma}^s, \pi_1(r)) \text{ with } R^s \sim_r R^c \end{array} \right\}$$

The proof of Theorem 1 is in Appendix D.

VII. Conclusions

Our computational soundness result bridges the gap between the cryptography community, where smart contracts have been investigated first, and the programming languages community. It guarantees that, if some safety properties are violated at the computational level, then they are also violated at the symbolic level. So, reachability-based symbolic analyses can be soundly used to prove safety properties of smart contracts.

BitML allows for writing contracts that never deadlock, e.g. the timed commitment protocol and the lottery in Section II. Intuitively, this symbolic property is preserved by the compiler, which requires all the needed signatures in the stipulation phase. We conjecture that this liveness property can be proved, using the same notion of coherence of Theorem 1. Further, we could strengthen Theorem 1 to take probabilities into account. More specifically, when in the symbolic model an event has probability p, then in the computational model the same event should have a probability $p \pm \varepsilon$ where ε is negligible.

A first proposal of an high-level language relying on Bitcoin is [31]. This language allows to model the updates of a state machine as affine logic propositions. Users can "run" this machine by putting transactions on the blockchain, with the guarantee that only the legit updates can be performed. Using a computational model similar to ours (including coherence) it would be possible to establish computational soundness also for [31]. A downside of [31] 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 in Example 4.

The expressivess of BitML could be improved by replacing the Bitcoin scripting language, using e.g. the approach of [32].

References

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997, http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548.

[2] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies," in *IEEE S & P*, 2015, pp. 104–121.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust (POST)*, ser. LNCS, vol. 10204. Springer, 2017, pp. 164–186. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-54455-6_8

[4] "Understanding the DAO attack," June 2016, http://www.coindesk.com/understanding-dao-hack-journalists/.

[5] "Parity Wallet security alert," July 2017, https://paritytech.io/blog/security-alert.html.

[6] "A Postmortem on the Parity Multi-Sig library self-destruct," November 2017, https://goo.gl/Kw3gXi.

[7] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Fair two-party computations via Bitcoin deposits," in *Financial Cryptography Workshops*, ser. LNCS, vol. 8438. Springer, 2014, pp. 105–121.

[8] I. Bentov and R. Kumaresan, "How to use Bitcoin to design fair protocols," in *CRYPTO*, ser. LNCS, vol. 8617. Springer, 2014, pp. 421–439.

[9] A. Miller and I. Bentov, "Zero-collateral lotteries in Bitcoin and Ethereum," in *EuroS&P Workshops*, 2017, pp. 4–13.

[10] M. Bartoletti and R. Zunino, "Constant-deposit multiparty lotteries on Bitcoin," in *Financial Cryptography Workshops*, 2017, also in IACR Cryptology ePrint Archive 955/2016.

[11] R. Kumaresan, T. Moran, and I. Bentov, "How to use Bitcoin to play decentralized poker," in *ACM CCS*, 2015, pp. 195–206.

[12] W. Banasik, S. Dziembowski, and D. Malinowski, "Efficient zero-knowledge contingent payments in cryptocurrencies without scripts," in *ESORICS*, ser. LNCS, vol. 9879. Springer, 2016, pp. 261–280.

[13] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí, "A fair protocol for data trading based on bitcoin transactions," *Future Generation Computer Systems*, 2017.

[14] G. Maxwell, "The first successful zero-knowledge contingent payment," https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/, 2016.

[15] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on Bitcoin," in *IEEE S & P*, 2014, pp. 443–458.

[16] R. Kumaresan and I. Bentov, "How to use Bitcoin to incentivize correct computations," in *ACM CCS*, 2014, pp. 30–41.

[17] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino, "SoK: unraveling Bitcoin smart contracts," 2018, IACR.

[18] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *J. Cryptology*, vol. 20, no. 3, p. 395, 2007.

[19] "Bitcoin developer guide - escrow and arbitration," https://bitcoin.org/en/developer-guide#escrow-and-arbitration.

[20] BitFury group, "Smart contracts on Bitcoin blockchain," 2015, http://bitfury.com/content/5-white-papers-research/contracts-1.1.1.pdf.

[21] D. Boneh and M. Naor, "Timed commitments," in *CRYPTO*, ser. LNCS, vol. 1880. Springer, 2000, pp. 236–254.

[22] P. F. Syverson, "Weakly secret bit commitment: Applications to lotteries and fair exchange," in *IEEE CSFW*, 1998, pp. 2–13.

[23] D. M. Goldschlag, S. G. Stubblebine, and P. F. Syverson, "Temporarily hidden bit commitment and lottery applications," *Int. J. Inf. Sec.*, vol. 9, no. 1, pp. 33–50, 2010.

[24] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on Bitcoin," *Commun. ACM*, vol. 59, no. 4, pp. 76–84, 2016.

[25] A. Back and I. Bentov, "Note on fair coin toss via Bitcoin," http://www.cs.technion.ac.il/~iddo/cointossBitcoin.pdf, 2013.

[26] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of Bitcoin transactions," in *Financial Cryptography and Data Security*, 2018.

[27] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *ACM Conference on Computer and Communications Security*. ACM, 1993, pp. 62–73.

[28] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," in *CRYPTO*, 2017, pp. 324–356.

[29] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin backbone protocol: Analysis and applications," in *EUROCRYPT*, ser. LNCS, vol. 9057. Springer, 2015, pp. 281–310.

[30] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE Symp. on Security and Privacy*, 2016, pp. 839–858.

[31] K. Crary and M. J. Sullivan, "Peer-to-peer affine commitment using Bitcoin," in *ACM Conf. on Programming Language Design and Implementation*, 2015, pp. 479–488.

[32] R. O'Connor, "Simplicity: A new language for blockchains," in *PLAS*, 2017. [Online]. Available: http://arxiv.org/abs/1711.03028

APPENDIX

A. Supplementary material for Section II

Definition 24 (Semantics of predicates).

$$
\begin{aligned}
[\![true]\!]_\Gamma &= true \\
[\![p_1 \wedge p_2]\!]_\Gamma &= [\![p_1]\!]_\Gamma \text{ and } [\![p_2]\!]_\Gamma \\
[\![\neg p]\!]_\Gamma &= \text{not } [\![p]\!]_\Gamma \\
[\![E_1 \circ E_2]\!]_\Gamma &= [\![E_1]\!]_\Gamma \circ [\![E_2]\!]_\Gamma \qquad \circ \in \{=, <\} \\
[\![N]\!]_\Gamma &= N \\
[\![|a|]\!]_\Gamma &= N \qquad \text{if } \Gamma \text{ contains } \mathsf{A}[a\#N] \\
[\![E_1 \bullet E_2]\!]_\Gamma &= [\![E_1]\!]_\Gamma \bullet [\![E_2]\!]_\Gamma \qquad \bullet \in \{+, -\}
\end{aligned}
$$

The above semantics is well-defined, provided that there exists a unique $\mathsf{A}[a\#N]$ in Γ for each $|a|$ in the predicate. Our semantics of contracts will ensure that is indeed the case.

We now describe the rules of deposits, starting from the simplest ones: those which deal with deposits, without directly affecting contracts.

The rule [Dep-AuthJoin] allows A to authorize the merge of two deposits x, y into a single one, creating the needed authorization. The label of the form $\mathsf{A} : \cdots$ records that only participant A can perform this move. The rule [Dep-Join] uses this authorization to create a single deposit z of A. The rules [Dep-AuthDivide] and [Dep-Divide] act similarly, allowing a deposit of A to be divided in two parts. The rules [Dep-AuthDonate] and [Dep-Donate] allow A to transfer one of her deposits to another participant. The pair of rules [Dep-AuthDestroy] and [Dep-Destroy] allow a set of participants to destroy a set of deposits $x_1 \cdots x_n$. To do that, first each participant A_i must provide the needed authorization $\mathsf{A}_i[\boldsymbol{x}, i \rhd y]$ for their own deposit x_i. When all the authorizations have been collected, rule [Dep-destroy] eliminates the deposits. The last pair of rules is needed to properly represent the fact that computational participants can create (and put on the ledger) transactions which do not have a counterpart in our symbolic model. To achieve a meaningful correspondence between the symbolic and the computational models, putting on the ledger such transactions is rendered with the rule [Dep-destroy].

B. Supplementary material for Section III

The following lemma states that stripping preserves the symbolic transitions, except for those which reveal secrets.

Lemma 2. *Let $\alpha \neq \mathsf{A} : a$, for any A and a. (i) if $R^s \xrightarrow{\alpha} \dot{R}^s$, then $strip(R^s) \xrightarrow{\alpha} strip(\dot{R}^s)$; (ii) if $strip(R^s) \xrightarrow{\alpha} R^s_*$, then $R^s \xrightarrow{\alpha} \dot{R}^s$, for some $\dot{R}^s$ such that $strip(\dot{R}^s) = strip(R^s_*)$.*

Lemma 3. *Let $R^s_* = strip(R^s)$ for some symbolic run R^s. Also, let $\mathsf{Hon} = \{\mathsf{A}_1, \ldots, \mathsf{A}_k\}$, and let $\boldsymbol{\Lambda^s} = \Lambda^s_1 \cdots \Lambda^s_k$, where $\Lambda^s_i = \Sigma^s_{\mathsf{A}_i}(R^s_*, r_{\mathsf{A}_i})$ for $i \in 1..k$. If $\Sigma^s_{\mathsf{Adv}}(R^s_*, r, \boldsymbol{\Lambda^s}) = \lambda^s$, then:*

1) *if $\lambda^s = (\mathsf{A}_i, j)$, then $R^s \xrightarrow{\alpha_j}$, where $\Lambda^s_i = \alpha_1 \cdots \alpha_m$;*
2) *otherwise, $R^s \xrightarrow{\lambda^s}$.*

$\mathsf{A}, \mathsf{B}, \ldots \in \mathsf{Part}$	Participants	$\Gamma, \Delta, \ldots$	Configuration (untimed)	r	Random generator
$x, y, \ldots$	Deposit names	(Γ, t)	Configuration (timed)	r_{A}	Random nonce (of A)
$a, a', b, \ldots$	Secrets names	$\alpha, \alpha', \ldots$	Labels (including delays)	$m, m', \ldots$	Bitstrings
$t, t', \delta, \ldots \in \mathbb{N}$	Delays	$R^s, \dot{R}^s, \ldots$	Symbolic runs	$\mathsf{T}, \mathsf{T}', \ldots$	Transactions
$v, v' \ldots \in 10^{-8}\mathbb{N}$	Currency values	$R^s_*, \dot{R}^s_*, \ldots$	Stripped symb. runs	$R^c, \dot{R}^c, \ldots$	Computational runs
$G, G', \ldots$	Preconditions	λ^s	Symbolic label	$R^c_*, \dot{R}^c_*, \ldots$	Stripped comp. runs
$C, C', \ldots$	Contracts	Λ^s	Set of symbolic labels	λ^c	Computational label
$D, D', \ldots$	Guarded contracts	Σ^s_{A}	Symbolic strategy	Λ^c	Set of computational labels
$\{G\}C$	Advertisement	$\mathfrak{B}_{\mathbf{adv}}(\cdot)$	Compiler	Σ^c_{A}	Computational strategy
				$\aleph(\cdot)$	Strategy mapping

TABLE I: Summary of notation.

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \xrightarrow{\mathsf{A}:x,y} \langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v+v'\rangle] \mid \Gamma}\ [\textsc{Dep-AuthJoin}]$$

$$\frac{\Gamma = \mathsf{A}[x, y \rhd \langle \mathsf{A}, v+v'\rangle] \mid \mathsf{A}[y, x \rhd \langle \mathsf{A}, v+v'\rangle] \mid \Gamma' \quad z \text{ fresh}}{\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \xrightarrow{join(x,y)} \langle \mathsf{A}, v+v'\rangle_z \mid \Gamma}\ [\textsc{Dep-Join}]$$

$$\frac{}{\langle \mathsf{A}, v+v'\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,v,v'} \langle \mathsf{A}, v+v'\rangle_x \mid \mathsf{A}[x \rhd \langle \mathsf{A}, v\rangle, \langle \mathsf{A}, v'\rangle] \mid \Gamma}\ [\textsc{Dep-AuthDivide}]$$

$$\frac{\Gamma = \mathsf{A}[x \rhd \langle \mathsf{A}, v\rangle, \langle \mathsf{A}, v'\rangle] \mid \Gamma' \quad y, y' \text{ fresh}}{\langle \mathsf{A}, v+v'\rangle_x \mid \Gamma \xrightarrow{divide(x,v,v')} \langle \mathsf{A}, v\rangle_y \mid \langle \mathsf{A}, v'\rangle_{y'} \mid \Gamma}\ [\textsc{Dep-Divide}]$$

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,\mathsf{B}} \langle \mathsf{A}, v\rangle_x \mid \mathsf{A}[x \rhd \mathsf{B}] \mid \Gamma}\ [\textsc{Dep-AuthDonate}] \qquad \frac{\Gamma = \mathsf{A}[x \rhd \mathsf{B}] \mid \Gamma' \quad y \text{ fresh}}{\langle \mathsf{A}, v\rangle_x \mid \Gamma \xrightarrow{donate(x,\mathsf{B})} \langle \mathsf{B}, v\rangle_y \mid \Gamma}\ [\textsc{Dep-Donate}]$$

$$\frac{\boldsymbol{x} = x_1 \cdots x_n \quad j \in 1..n \quad y \text{ fresh (except in destroy authorizations for } \boldsymbol{x})}{\left(\|_{i\in 1..n} \langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \Gamma \xrightarrow{\mathsf{A}_j:\boldsymbol{x},j} \left(\|_{i\in 1..n} \langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \mathsf{A}_j[\boldsymbol{x}, j \rhd y] \mid \Gamma}\ [\textsc{Dep-AuthDestroy}]$$

$$\frac{\boldsymbol{x} = x_1 \cdots x_n \quad \Gamma = \left(\|_{i\in 1..n} \mathsf{A}_i[\boldsymbol{x}, i \rhd y]\right) \mid \Gamma'}{\left(\|_{i\in 1..n} \langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \Gamma \xrightarrow{destroy(\boldsymbol{x})} \Gamma}\ [\textsc{Dep-Destroy}]$$

Fig. 8: Semantics of untimed configurations: rules for deposits.

C. Supplementary material for Section IV

In Lemma 4 we will show that there exists exactly one such maximal run.

Lemma 4. *Let $\boldsymbol{\Sigma}^c$ be a set of computational strategies for the honest participants and for the adversary, and let r be a randomness source. There exist exactly one maximal run R^c which conforms to $(\boldsymbol{\Sigma}^c, r)$.*

D. Supplementary material for Section VI

Hereafter, we assume that at least one of the participants in $\{G\}C$ has a secret a in G. Since rule [C-Advertise] requires that secret names are fresh, this ensures that the same contract can not be advertised twice. Expressiveness is not affected by this assumption, since a needs not be used in C. In all our examples, we implicitly assume this condition to be respected.

Definition 25 (Coherence). We inductively define the relation $coher(R^s, R^c, r, txout, sechash, \kappa)$, where (i) R^s is a symbolic run, (ii) R^c is a computational run, (iii) r is a randomness source, (iv) $txout$ is an injective function from names x (occurring in R^s) to transaction ouputs (T, o) (where T occurs in R^c), respecting values; (v) $sechash$ is a mapping from secret names a (occurring in R^s) to bitstrings; (vi) κ maps triples $(\{G\}C, D, \mathsf{A})$, where D is a subterm of C, to public keys.

Base case: $coher(R^s, R^c, r, txout, sechash, \kappa)$ holds if all the following conditions hold: (i) $R^s = (\Gamma_0, 0)$, with Γ_0 initial; (ii) $R^c = \mathsf{T}_0 \cdots$ initial; (iii) all the public keys in R^c are generated from r, according to Definition 15; (iv) $txout$ maps exactly the x of $\langle \mathsf{A}, v\rangle_x$ in Γ_0 to an output in T_0 of value $v\mathfrak{B}$, and spendable with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$; (v) $\operatorname{dom} sechash = \emptyset$; (vi) $\operatorname{dom} \kappa = \emptyset$.

Inductive case: $coher(\dot{R}^s \xrightarrow{\alpha} (\Gamma, t), \dot{R}^c \lambda^c, r, txout, sechash, \kappa)$ holds if $coher(\dot{R}^s, \dot{R}^c, r, txout', sechash', \kappa')$ and one of the following cases applies.

1) $\alpha = advertise(\{G\}C)$, $\lambda^c = \mathsf{A} \to * : \mathcal{C}$, where $\mathcal{C}$ is obtained by encoding $\{G\}C$ as a bitstring, representing each x in it as the transaction output $txout'(x)$. Further, $txout' = txout$, $sechash' = sechash$, and $\kappa' = \kappa$.

2) $\alpha = \mathsf{A} : \{G\}C, \Delta$, where: (i) for some B, $\dot{R}^c$ contains $\mathsf{B} \to * : \mathcal{C}$, where $\mathcal{C}$ is obtained from $\{G\}C$ and $txout'$ as in Item 1. Note that $\dot{R}^c$ might contain several such messages; below, we let $\mathcal{C}$ represent the first occurrence. (ii) for some B, $\lambda^c = \mathsf{B} \to * : (\mathcal{C}, \boldsymbol{h}, \boldsymbol{k})$ (signed by A), where $\boldsymbol{h}$ is a sequence comprising a bitstring h_i with $|h_i| = \eta$ for each secret a_i in Δ, and $\boldsymbol{k}$ is a sequence of keys, as the one produced by the stipulation protocol. We require that λ^c is the first occurrence, in the run $\dot{R}^c$, of such a message after $\mathcal{C}$. (iii) Let N_i be the length of a_i fixed in Δ. If $N_i \neq \bot$, we require that $\dot{R}^c$ contains, for some B, a query to the oracle $\mathsf{B} \to \mathsf{O} : m_i$, and a subsequent reply $\mathsf{O} \to \mathsf{B} : h_i$ such that $|m_i| = \eta + N_i$. Otherwise, if $N_i = \bot$, we require that h_i does not occur as a reply from O to any query of length $\geq \eta$. (iv) No hash is reused: the h_i are pairwise distinct, and also distinct from $sechash'(b)$ for any $b \in \mathrm{dom}\,(sechash')$. (v) $txout = txout'$. (vi) $sechash$ extends $sechash'$ so that for each secret a_i we have $sechash(a_i) = h_i$. (vii) If $\mathsf{A} \in \mathsf{Hon}$, we define κ by extending κ' according to $\boldsymbol{k}$, so to record the public keys of all participants occurring in G for each subterm D of C. If κ' already defines such keys, or $\mathsf{A} \notin \mathsf{Hon}$, we let $\kappa = \kappa'$.

3) $\alpha = \mathsf{A} : \{G\}C, x$, where: (i) $\lambda^c = \mathsf{B} \to * : m$ for some B, where m is the signature of the transaction T_{init} of $\ddot{\mathsf{B}}_{\mathbf{adv}}(\{G\}C)$ relatively to the input x with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. The parameters of the compiler are set as follows: $\mathbf{part}$, PartG and $\mathbf{val}$ are inferred from G, we let $\mathbf{txout} = txout'$, $\mathbf{sechash} = sechash'$, and $\mathbf{K}(\mathsf{B}) = \hat{K}^p_{\mathsf{B}}(r_{\mathsf{B}})$, $\mathbf{K}(D, \mathsf{B}) = \kappa'(\{G\}C, D, \mathsf{B})$ for each B, and D subterm of C. (ii) For some B, we have $\mathsf{B} \to * : \mathsf{T}_{init}$ occurring in $\dot{R}^c$. (iii) λ^c is the first occurrence in $\dot{R}^c$ of a broadcast of m after T_{init}. (iv) $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$.

4) $\alpha = init(G, C)$, where: (i) α consumes from $\dot{R}^s$ the advertisement $\{G\}C$ and its persistent deposits to produce $\langle C, v\rangle_z$. (ii) $\lambda^c = \mathsf{T}_{init}$ where T_{init} is the first transaction in $\ddot{\mathsf{B}}_{\mathbf{adv}}(\{G\}C)$. The needed compiler parameters are obtained as in Item 3. (iii) $sechash = sechash'$, $\kappa = \kappa'$, and $txout$ extends $txout'$, mapping z to T_{init}.

5) $\alpha = \mathsf{A} : x, D$, where: (i) $\dot{R}^s$ contains $\langle C', v\rangle_x$ with $C' = D + \sum_i D_i$, for some $D = \mathsf{A} : D'$. (ii) In $\dot{R}^s$, we find that $\langle C', v\rangle_x$ has $\{G\}C$ as its ancestor advertisement. (iii) $\lambda^c = \mathsf{B} \to * : m$, where m is a signature with key $\kappa'(\{G\}C, D, \mathsf{A})$ of the first transaction T in $\ddot{\mathsf{B}}_{\mathbf{D}}(D, D, \mathsf{T}', o, v, \mathsf{PartG}, 0)$, where $(\mathsf{T}', o) = txout'(x)$. The compiler parameters are obtained as in Item 3. (iv) $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$. (v) $\dot{R}^c$ contains $\mathsf{B} \to * : \mathsf{T}$ for some B, and m is the first signature of T in $\dot{R}^c\lambda^c$ after the first broadcast of T.

6) $\alpha = put(\boldsymbol{x}, \boldsymbol{a}, y)$, where: (i) $\boldsymbol{x} = x_1 \cdots x_k$. (ii) In $\Gamma_{\dot{R}^s}$, the action α consumes $\langle D + C, v\rangle_y$ and the deposits $\langle \mathsf{A}_i, v_i\rangle_{x_i}$ to produce $\langle C', v'\rangle_{y'}$, where $D = \cdots : \mathtt{put} \cdots \mathtt{reveal} \cdots . C'$. Let t be maximum deadline in an $\mathtt{after}$ in front of D. (iii) In $\dot{R}^s$, we find that $\langle D + C, v\rangle_y$ has $\{G\}C''$ as its ancestor advertisement, for some G and C''. (iv) $\lambda^c = \mathsf{T}$ where T is the first transaction of $\ddot{\mathsf{B}}_{\mathbf{C}}(C', D, \mathsf{T}', o, v', \boldsymbol{x}, \mathsf{PartG}, t)$, where $(\mathsf{T}', o) = txout'(y)$. The compiler parameters are obtained as in Item 3. (v) $txout$ extends $txout'$ so that y' is mapped to $(\mathsf{T}, 0)$, $sechash = sechash'$, and $\kappa = \kappa'$.

7) $\alpha = \mathsf{A} : a$, where: (i) $\lambda^c = \mathsf{B} \to * : m$ from some B with $|m| \geq \eta$. (ii) $\dot{R}^c = \cdots (\mathsf{B} \to \mathsf{O} : m)(\mathsf{O} \to \mathsf{B} : sechash'(a)) \cdots$, for some B. (iii) $txout = txout'$, $sechash = sechash'$ and $\kappa = \kappa'$. (iv) In $\dot{R}^s$ we find an $\mathsf{A} : \{G\}C, \Delta$ action, with a in G, with a corresponding broadcast in $\dot{R}^c$ of $m' = (\mathcal{C}, \boldsymbol{h}, \boldsymbol{k})$. (v) λ^c is the first broadcast of m in $\dot{R}^c$ after the first broadcast of m'.

8) $\alpha = split(y)$, where: (i) In $\dot{R}^s$, the action α consumes $\langle D + C, v\rangle_y$ to obtain $\langle C_0, v_0\rangle_{x_0} \mid \cdots \mid \langle C_k, v_k\rangle_{x_k}$ where $D = \cdots : \mathtt{split}\ \boldsymbol{v} \to \boldsymbol{C}$ and $\boldsymbol{C} = C_0 \ldots C_k$. Let t be the maximum deadline in an $\mathtt{after}$ in front of D. (ii) In $\dot{R}^s$, we find that $\langle D + C, v\rangle_y$ has $\{G\}C'$ as its ancestor advertisement. (iii) $\lambda^c = \mathsf{T}$ where T is the first transaction of $\ddot{\mathsf{B}}_{\mathbf{par}}(\boldsymbol{C}, D, \mathsf{T}', o, \mathsf{PartG}, t)$ where $(\mathsf{T}', o) = txout'(y)$. The compiler parameters are obtained as for Item 3. (iv) $txout$ extends $txout'$ mapping each x_i to (T, i), $sechash = sechash'$, and $\kappa = \kappa'$.

9) $\alpha = withdraw(\mathsf{A}, v, y)$, where: (i) In $\dot{R}^s$, the action α consumes $\langle D + C, v\rangle_y$ to obtain $\langle \mathsf{A}, v\rangle_x$, where $D = \cdots : \mathtt{withdraw}\ \mathsf{A}$. (ii) In $\dot{R}^s$, we find that $\langle D + C, v\rangle_y$ has $\{G\}C'$ as its ancestor advertisement. (iii) $\lambda^c = \mathsf{T}$ where T is the first transaction of $\ddot{\mathsf{B}}_{\mathbf{D}}(D, D, \mathsf{T}', o, v, \mathsf{PartG}, 0)$ where $(\mathsf{T}', o) = txout'(y)$. The compiler parameters are obtained as for Item 3. (iv) $txout$ extends $txout'$ mapping x to $(\mathsf{T}, 0)$, $sechash = sechash'$, and $\kappa = \kappa'$.

10) $\alpha = \mathsf{A} : x, x'$, where: (i) In $\dot{R}^s$ we find $\langle \mathsf{A}, v\rangle_x$ and $\langle \mathsf{A}, v'\rangle_{x'}$. (ii) In $\dot{R}^c$ we find $\mathsf{B} \to * : \mathsf{T}$ for some B, T, where T has as its two inputs $txout'(x)$ and $txout'(x')$, and a single output of value $v + v'$ redeemable with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. (iii) $\lambda^c = \mathsf{B} \to * : m'$ for some B, m', where m' is the signature of T with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. (iv) λ^c is the first broadcast of m' in $\dot{R}^c$ after the first broadcast of T. (v) $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$.

11) $\alpha = join(x, y)$, where: (i) In $\dot{R}^s$ the action α spends $\langle \mathsf{A}, v\rangle_x$ and $\langle \mathsf{A}, v'\rangle_{x'}$ to obtain $\langle \mathsf{A}, v + v'\rangle_y$. (ii) $\lambda^c = \mathsf{T}$ is a transaction having as inputs $txout'(x)$ and $txout'(x')$, and having one output of value $v + v'$ redeemable with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$. (iii) $txout$ extends $txout'$ mapping y to $(\mathsf{T}, 0)$, $sechash = sechash'$, and $\kappa = \kappa'$.

12) $\alpha = \mathsf{A} : x, v, v'$. Similar to Item 10.

13) $\alpha = divide(x, v, v')$. Similar to Item 11.

14) $\alpha = \mathsf{A} : x, \mathsf{B}$. Similar to Item 10.

15) $\alpha = donate(x, \mathsf{B})$. Similar to Item 11.

16) $\alpha = \mathsf{A} : \boldsymbol{y}, j$, where: (i) $\boldsymbol{y} = y_1 \cdots y_k$. (ii) In $\dot{R}^s$ we find $\langle \mathsf{B}_i, v_i\rangle_{y_i}$ for $i \in 1..k$, with $\mathsf{B}_j = \mathsf{A}$. (iii) In $\dot{R}^c$ we find $\mathsf{B} \to * : \mathsf{T}$ for some B, T, where T has as its inputs $txout'(y_i)$ for $i \in 1..k$, and possibly others not in $\mathrm{ran}\, txout'$. (iv) $\lambda^c = \mathsf{B} \to * : m$ from some B, m where m is a signature of T with $\hat{K}_{\mathsf{A}}(r_{\mathsf{A}})$, corresponding to

the j-th input. (v) λ^c is the first broadcast of m in $\dot{R}^c$ after the first broadcast of T. (vi) λ^c does not correspond to any of the other cases, i.e. there is no other symbolic action α for which $\dot{R}^s\alpha$ would be coherent with $\dot{R}^c\lambda^c$. (vii) $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$.

17) $\alpha = destroy(\boldsymbol{x})$, where: (i) $\boldsymbol{x} = x_1 \cdots x_k$. (ii) In $\dot{R}^s$, α consumes $\langle \mathsf{A}_i, v_i \rangle_{x_i}$ to obtain 0. (iii) $\lambda^c = \mathsf{T}$ from some T having as inputs $txout'(x_1), \ldots, txout'(x_k)$, and possibly others not in $\operatorname{ran} txout'$. (iv) λ^c does not correspond to any of the other cases, i.e. there is no other symbolic action α for which $\dot{R}^s\alpha$ would be coherent with $\dot{R}^c\lambda^c$. (v) $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$.

18) $\alpha = \delta = \lambda^c$, and $txout = txout'$, $sechash = sechash'$, and $\kappa = \kappa'$.

Inductive case 2: $coher(R^s, R^c\lambda^c, r, txout, sechash, \kappa)$ holds if $coher(R^s, R^c, r, txout, sechash, \kappa)$ and one of the following cases applies:

1) $\lambda^c = \mathsf{T}$ where no input of T belongs to $\operatorname{ran} txout$.
2) $\lambda^c = \mathsf{A} \to \mathsf{O} : m$ or $\lambda^c = \mathsf{O} \to \mathsf{A} : m$, for some A, m.
3) $\lambda^c = \mathsf{A} \to * : m$, where λ^c does not correspond to any symbolic move, according to the first inductive case.

We write $R^s \sim_r R^c$ iff $coher(R^s, R^c, r, txout, sechash, \kappa)$ for some $txout$, $sechash$, and κ. $\diamond$

Proof of Theorem 1 (sketch): Assume that R^c satisfies the hypotheses, but has no corresponding R^s which is coherent (to R^c) and conforming (to the symbolic strategies). Consider the longest prefix $\dot{R}^c$ of R^c having a corresponding $\dot{R}^s$ which is coherent (to $\dot{R}^c$) and conforming (to the computational strategies). We have that $R^c = \dot{R}^c\lambda^c \cdots$. We now show that either $\dot{R}^c\lambda^c$ has a corresponding symbolic run $\dot{R}^s\ddot{R}^s$ which is coherent and conforming to the symbolic strategies (contradicting the maximality of $\dot{R}^c$), or the adversary succeeded in a signature forgery, or in a preimage attack (which can happen only with negligible probability). Note that, by obtain coherence, $\ddot{R}^s$ must be either empty, or contain a single symbolic action.

We proceed by cases on λ^c:

1) $\lambda^c = \mathsf{B} \to * : m$. Then, coherence must hold for some $\ddot{R}^s$. Indeed, the definition of coherence maps m to an authorization (if it is the first broadcast of a signature), a revealed secret (if it is the first broadcast of a preimage), or in *all* other cases it simply ignores m. So, we can choose $\ddot{R}^s$ as the corresponding move, or to be empty, and obtain coherence. In these cases, we also obtain conformance. Indeed, in the last case ($\ddot{R}^s$ empty) the run $\dot{R}^s\ddot{R}^s = \dot{R}^s$ is trivially conforming. For the authorization or reveal cases, we note that if the computational Adv was able to generate m, it is either forged (with negligible probability), or it originated from some honest A. Since $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$, it follows that, at some time in the past, Σ^s_{A} enabled the authorization or reveal. By persistency, it is also enabled at the end of $\dot{R}^s$, hence Σ^s_{Adv} can choose such action, and achieve conformance.

2) If $\lambda^c = \mathsf{T}$, we consider the following subcases according to the inputs of T:
   a) If no input of T belongs to $\operatorname{ran} txout$, then coherence and conformance are achieved taking $\ddot{R}^s$ to be empty.
   b) Otherwise, if at least one of the inputs of T belongs to $\operatorname{ran} txout$, then we look in $\dot{R}^s$ for all the deposits and active contracts corresponding to such inputs. By definition of computational strategy, we must find in $\dot{R}^c$ a (first) broadcast $\mathsf{B} \to * : \mathsf{T}$ followed by a (first) broadcast $\mathsf{B} \to * : m$ for all witnesses m of T. By the coherence of $\dot{R}^c$, in $\dot{R}^s$ the messages $\mathsf{B} \to * : m$ correspond to suitable authorization/reveal moves for each of the (counterparts of the) inputs of T. We consider the following subcases:
      i) If all the inputs are deposits, then we let $\ddot{R}^s$ perform the symbolic move corresponding to T (e.g., init or join). Note that if T can not be represented symbolically, we can choose $\ddot{R}^s$ to perform a destroy. Such moves are feasible symbolically since we already have their authorizations. Such $\ddot{R}^s$ leads to a coherent run, which is also conforming, since even if no honest strategy wants to perform the move, Adv can perform it on its own, having all the authorizations.
      ii) Otherwise, some input T' of T must correspond to an active contract. This must be originated from an advertisement, which has to involve at least one honest participant A, by definition of the symbolic semantics (rule [C-Advertise]). However, by construction, our compiler makes T' require the signatures of all the participants, hence including A. Since such signature must occur as a witness in T, the adversary Adv must have forged it (with negligible probability), or must have obtained it from A. In the latter case, $\dot{R}^s$ contains an authorization for the symbolic move corresponding to λ^c. By choosing $\ddot{R}^s$ accordingly, we obtain a coherent and conforming run.
3) Finally, if $\lambda^c = \delta$, we simply choose $\ddot{R}^s$ to perform δ. Coherence trivially holds. For conformance, we note that by definition of computational strategy, all the honest participants must output a Λ^c which either contains some $\delta' \geq \delta$, or is empty. By definition of $\aleph$, this must also be the case in Λ^s, resulting in conformance. $\square$