Secure compilation of rich smart contracts on poor UTXO blockchains

Massimo Bartoletti
Università degli Studi di Cagliari
Cagliari, Italy
bart@unica.it

Riccardo Marchesin
Università degli Studi di Trento
Trento, Italy
riccardo.marchesin@unitn.it

Roberto Zunino
Università degli Studi di Trento
Trento, Italy
roberto.zunino@unitn.it

***Abstract*—Most blockchain platforms from Ethereum onwards render smart contracts as stateful reactive objects that update their state and transfer crypto-assets in response to transactions. A drawback of this design is that when users submit a transaction, they cannot predict in which state it will be executed. This exposes them to transaction-ordering attacks, a widespread class of attacks where adversaries with the power to construct blocks of transactions can extract value from smart contracts (the so-called MEV attacks). The UTXO model is an alternative blockchain design that thwarts these attacks by requiring new transactions to spend past ones: since transactions have unique identifiers, reordering attacks are ineffective. Currently, the blockchains following the UTXO model either provide contracts with limited expressiveness (Bitcoin), or require complex run-time environments (Cardano). We present ILLUM, an Intermediate-Level Language for the UTXO Model. ILLUM can express real-world smart contracts, e.g. those found in Decentralized Finance. We define a compiler from ILLUM to a bare-bone UTXO blockchain with loop-free scripts. Our compilation target only requires minimal extensions to Bitcoin Script: in particular, we exploit covenants, a mechanism for preserving scripts along chains of transactions. We prove the security of our compiler: namely, any attack targeting the compiled contract is also observable at the ILLUM level. Hence, the compiler does not introduce new vulnerabilities that were not already present in the source ILLUM contract. We evaluate the practicality of ILLUM as a compilation target for higher-level languages. To this purpose, we implement a compiler from a contract language inspired by Solidity to ILLUM, and we apply it to a benchmark or real-world smart contracts.**

***Index Terms*—Blockchain, smart contracts, UTXO model**

1. Introduction

Smart contracts are agreements between mutually untrusted parties that are enforceable by a computer program, without the need of a trusted intermediary. Currently, most implementations of smart contracts are based on permissionless blockchains, where the conjunction with crypto-assets has given rise to new applications, like decentralized finance (DeFi) [1] and decentralized autonomous organizations (DAOs) [2], that overall control nearly 90 billion dollars worth of assets today [3].

Two main smart contracts models have emerged so far. In the *account-based model*, contracts are reactive objects that live on the blockchain and process user transactions by updating their state and transferring crypto-assets among users [4]. In the *UTXO model*, instead, contracts, their state, and the ownership of assets are encoded within transactions: when a new transaction is published in the blockchain, it replaces ("spends") an old transaction, effectively updating the contract state and the assets ownership. The UTXO model was first proposed by Bitcoin, where the idea of blockchain-based contracts originated in 2012. The account-based model was later introduced in 2015 by Ethereum, where contracts were popularized. Most blockchain platforms today follow the account-based model: besides Ethereum, also other mainstream blockchains such as Solana, Avalanche, Hedera, Algorand and Tezos are account-based (albeit with differences, sometimes notable, from case to case).

Account-based vs. UTXO blockchains. In the account-based model, contracts can be seen as objects with a state accessible and modifiable by methods, as in object-oriented programming. For instance, to withdraw 10 token units from a `Bank` contract, a user `A` sends a transaction `withdraw(10)` to `Bank`, which will react by updating its state and `A`'s wallet. Programming contracts in the UTXO model requires instead a paradigm shift from the common object-oriented style [5]. Indeed, a UTXO transaction does not directly represent a contract action: rather, it encodes a transfer of crypto-assets from its *inputs* to its *outputs*. Transaction outputs specify the assets they control, the contract state, and the conditions under which the assets can be transferred again. Transaction inputs are references to unspent outputs of previous transactions, and provide the values that make their spending conditions true. The blockchain state is given by the set of Unspent Transaction Outputs (UTXO). A transaction can spend one or more of outputs in the UTXO set, specifying them as its inputs: this effectively removes these outputs from the blockchain state, and creates new ones. The new outputs update the state of the contracts, and redistribute the assets according to their spending conditions. These conditions are specified in a *scripting language*, the expressiveness of which is reflected on that of contracts. For instance, in the banking use case above, the state of the `Bank` contract could be scattered among a set of outputs. To withdraw, `A` must send a transaction which spends one or more of these outputs, and whose output has a spending condition that can be satisfied only by `A` (e.g., a signature verification against `A`'s public key). In addition, the `Bank` state in the new output must be a correct update of the old state (e.g., in the new state `A`'s account must have 10 tokens less than in the old state). Programming contracts in this model is more complex than in the account-based model, since the links to familiar programming abstractions are weaker.

Despite this additional complexity, the UTXO model has a series of advantages over the account-based model. A first problem of account-based stateful platforms like Ethereum is to undermine the concurrent execution of transactions. Namely, a multi-core blockchain node cannot simply execute transactions in parallel, since they may perform conflicting accesses to shared parts of the state, possibly leading to an inconsistent state [6]. In such platforms, there is no efficient way to detect when transactions can be safely parallelized: in general, determining the accessed parts of the state requires to fully execute them. In the UTXO model, instead, it is easy to detect when transactions are parallelizable: just check if they spend disjoint outputs, which can be done efficiently [7].

Another problem of the account-based model is that a user sending a transaction to the blockchain network cannot accurately predict the state in which it will be executed. This has several negative consequences, such as the unpredictability of transaction fees and the susceptibility to maximal extractable value (MEV) attacks [8]–[10]. Fees are a common incentive mechanism for the blockchain network to execute transactions and a defence against denial-of-service attacks. To be accepted, a transaction must pay a fee which is proportional to the computational resources needed to validate it. The actual amount of these resources heavily depends on the initial state where the transaction is performed: so, to be sure that their transactions are accepted, users specify a maximum fee they are willing to pay. Besides forcing users to over-approximate fees, this also opens to attacks where the adversary front-runs transactions so that they are executed in a state where the paid fee is insufficient: the consequence is that users pay the fee even for rejected transactions that do not update the contract state according to their intention. With MEV attacks instead, the adversary colludes with malicious blockchain nodes to propose blocks where the ordering of transactions is profitable for the adversary (to the detriment of users). These attacks are very common in account-based blockchains (targeting in particular DeFi contracts), and are estimated to be worth more than USD 1 billion [11] so far.

The UTXO model naturally mitigates these attacks. Indeed, when a user sends a transaction T to the blockchain network, they know *exactly* in which state it will be executed, since this state is completely determined by T's inputs. Therefore, if an adversary M front-runs T with their transaction T_M, the transaction T will be rejected by the blockchain network, since some of its inputs are spent by T_M. If the user still desires to perform the action in the new state, they must resend T, updating its inputs (and therefore, specifying the new state where the action is executed). This thwarts both the fees exhaustion attacks and the MEV attacks described before.

UTXO designs: Bitcoin vs. Cardano. Currently, the two main UTXO blockchains are Bitcoin and Cardano. These platforms follow radically different design choices in the structure of transactions and in the scripting languages to specify their spending conditions. These differences deeply affect the expressiveness of their contracts and the complexity of their runtime environments. On the one hand, Bitcoin has a minimal scripting language, featuring only basic arithmetic and logical operations, conditionals, hashes, and (limited) signature verification [12]. This imposes a stringent limit on the expressiveness of contracts in Bitcoin: contracts requiring unbounded computational steps, or transfers of tokens different than native crypto-currency, cannot be expressed [13]. Neglecting the lack of expressiveness, the design choice of keeping the scripting language minimal has some positive aspects: besides limiting the attack surface and simplifying the overall design (e.g., no gas mechanism is needed), it facilitates the formal verification of contracts. On the other side of the spectrum, Cardano's scripting language is an untyped lambda-calculus [14], which makes Cardano scripts, and in turn contracts, Turing-complete. This increase in expressiveness comes at a cost, in that the static verification of general contract properties is undecidable. Furthermore, since Cardano's scripts feature unbounded iteration, a gas mechanism is needed to abort time-consuming computations and suitably reward blockchain nodes for validating transactions. Although the gas needed to execute a transaction is statically known (unlike in account-based blockchains, where it depends on the actual state where the transaction is executed), still it would be safer to avoid the gas mechanism altogether. For instance, a misalignment of the gas incentives led to DoS attack on Ethereum [15]. Another drawback of Bitcoin, Cardano, and UTXO blockchains in general, is that, due to the absence of explicit state, writing stateful contracts is more difficult than in account-based blockchains [5].

Our research question is whether one can find a balance between the two approaches which also overcomes their usability issues. More specifically, ours is a quest for a contract language and a UXTO model such that:

- the expressiveness of contracts is enough for real-world use cases (contracts are Turing-complete);
- executing contracts requires a simple blockchain design (individual transaction scripts are not Turing-complete, and no gas mechanism is required);
- it can serve as a compilation target of developer-friendly higher-level contract languages.

Contributions. We address this research question by proposing an expressive intermediate-level contract language that compiles into transactions executable by a bare-bone UTXO blockchain (with no gas mechanism). The key insight is to scatter the execution of complex contract actions across multiple UTXO transactions. Even if each of these transactions contains only simple (loop-free) scripts, the overall chain of transactions can encompass complex (possibly recursive) behaviours.

We summarize our main contributions as follows:

- ILLUM, an Intermediate Level Language for UTXO blockchains. ILLUM is a Turing-complete clause language with primitives to exchange crypto-assets. We evaluate ILLUM on a few use cases, including gambling games, auctions and Ponzi schemes (Section 2).
- a compiler from ILLUM to UTXO transactions. The scripting language used in these transactions is Bitcoin Script extended with *covenants*, operators to constrain the output scripts of the redeeming transactions [16]. This is a lightweight mechanism, which can be implemented with minimal overhead on the runtime of UTXO blockchains [17], [18].

- a proof of the security of the ILLUM compiler. Namely, we prove that, even in the presence of adversaries, with overwhelming probability there is a step-by-step correspondence between the execution of an ILLUM contract and that of the chain of transactions resulting from its compilation. Our security result essentially establishes Robust Trace Property Preservation [19], [20], ensuring that each computational trace (involving any computational adversary) has a symbolic counterpart (involving a suitable symbolic adversary). Its proof is quite complex, as it matches every possible contract action in ILLUM (more than 20 cases) with some action at the blockchain level.
- a prototype implementation of a compiler from a Solidity-like high-level contract language to ILLUM. We illustrate our compilation technique in Section 7.
- an evaluation of the practicality of our approach, based on a benchmark of common smart contracts that we implement in the high-level language and then translate to ILLUM with our prototype compiler. Overall, our evaluation shows that it is feasible to reconcile the UTXO model with the familiar procedural programming style supported by Solidity, effectively making UTXO contracts more usable in practice.

Because of space constraints, we refer to a technical report for the proofs [21], and to a github repository[1] for the code of the prototype compiler and of the benchmark.

2. Overview

In this section we overview our approach, discussing its main features and results. Here we will mostly focus on intuition, leveraging on examples and postponing the full technical development to later sections.

2.1. An intermediate contract language

ILLUM is a clause-based process calculus that can serve as an intermediate contract language and compiles to a bare-bone UTXO model. ILLUM contracts are sets of ***clauses***, each having a defining equation of the form:

$$\mathtt{X}(\boldsymbol{In}\,;\boldsymbol{Ex}) = \{\boldsymbol{v}\mathrm{T}\ \ \mathtt{if}\ \ p\}\ C$$

Here, $\mathtt{X}$ is the clause name, $\boldsymbol{In}$ and $\boldsymbol{Ex}$ are sequences of formal parameters (respectively, *internal* and *external*), $\{\boldsymbol{v}\mathrm{T}\ \ \mathtt{if}\ \ p\}$ is the *funding precondition* (namely: "$\boldsymbol{v}$ units of tokens T are available and the condition p is true"), and C is a *process* encoding the clause behaviour. We provide some intuition about these constructs with an example.

Example: a "double or nothing" game. We specify a gambling game between two players A and B as follows:

$$\begin{aligned}
\mathtt{Init}(;) &= \{0\}\ (\mathtt{call}\ \mathtt{Play}_{\mathsf{A}}\langle 1;\rangle + \mathtt{call}\ \mathtt{Play}_{\mathsf{B}}\langle 1;\rangle)\\
\mathtt{Play}_{\mathsf{A}}(v;) &= \{v\mathrm{T}\}\ (\mathtt{call}\ \mathtt{Play}_{\mathsf{B}}\langle 2v;\rangle\\
&\qquad + \mathtt{afterRel}\ 5 : \mathtt{send}\ (v\mathrm{T} \to \mathsf{A}))\\
\mathtt{Play}_{\mathsf{B}}(v;) &= \{v\mathrm{T}\}\ (\mathtt{call}\ \mathtt{Play}_{\mathsf{A}}\langle 2v;\rangle\\
&\qquad + \mathtt{afterRel}\ 5 : \mathtt{send}\ (v\mathrm{T} \to \mathsf{B}))
\end{aligned}$$

Here, we have not used external parameters, and we have omitted writing $\mathtt{if}\ \mathit{true}$ in the funding precondition.

1. https://github.com/bitbart/illum-lang/

To start the game, participants can invoke the $\mathtt{Init}$ clause. Since it has no funding precondition, this does not require paying tokens upfront. After that, the contract gives two options: either calling $\mathtt{Play}_{\mathsf{A}}$ or $\mathtt{Play}_{\mathsf{B}}$. Choosing either option requires the player to satisfy its funding precondition: here, both clauses require paying $v = 1$ tokens T, since the internal parameter v is set to 1 by the caller $\mathtt{Init}$. Now, assume that $\mathtt{Play}_{\mathsf{A}}$ was chosen. The clause offers two new options: either calling $\mathtt{Play}_{\mathsf{B}}$ with a doubled internal parameter v, or sending v units of T to player A after 5 time units. The first option requires to pay other $v = 1$ tokens to the contract, so that its new balance satisfies the funding precondition of $\mathtt{Play}_{\mathsf{B}}$. After that, both players can take turns doubling the contract balance, until one of them fails to do so within 5 time units. When this happens, the other player can redeem the whole contract balance, ending the game.

In the game seen so far, the contract balance is fully determined by the contract itself, with players having no choice in regard to how many tokens they can add. External parameters allow us to make the game more interesting, letting players arbitrarily raise the bet as long as it is greater than the double of the previous balance:

$$\begin{aligned}
\mathtt{Play}_{\mathsf{A}}\langle v;w\rangle &= \{w\mathrm{T}\ \ \mathtt{if}\ \ w > 2v\}\ (\mathtt{call}\ \mathtt{Play}_{\mathsf{B}}\langle w;?\rangle\\
&\qquad + \mathtt{afterRel}\ 5 : \mathtt{send}\ (w\mathrm{T} \to \mathsf{A})\ )\\
\mathtt{Play}_{\mathsf{B}}\langle \bar{v};\bar{w}\rangle &= \{\bar{w}\mathrm{T}\ \ \mathtt{if}\ \ \bar{w} > 2\bar{v}\}\ (\mathtt{call}\ \mathtt{Play}_{\mathsf{A}}\langle \bar{w};?\rangle\\
&\qquad + \mathtt{afterRel}\ 5 : \mathtt{send}\ (\bar{w}\mathrm{T} \to \mathsf{B})\ )
\end{aligned}$$

For instance, assume that $\mathtt{Play}_{\mathsf{A}}\langle v;w\rangle$ is active when a player executes $\mathtt{call}\ \mathtt{Play}_{\mathsf{B}}\langle w;?\rangle$. The internal parameter $\bar{v} = w$ is determined by the process, while the external parameter $\bar{w} =?$ is chosen by the player, provided that it respects the funding precondition of $\mathtt{Play}_{\mathsf{B}}$, namely $\bar{w} > 2\bar{v}$. This means that the player must pay enough tokens to make the contract balance reach $2w$ tokens, doubling the previous balance w.

More on ILLUM clauses. Generalising from the previous examples, processes C are choices $D_1 + \cdots + D_k$ among one or more branches. Branches have two forms:

- $\mathtt{send}\ (v_1\mathrm{T}_1 \to \mathsf{A}_1 \| \cdots \| v_n\mathrm{T}_n \to \mathsf{A}_n)$. This branch sends v_i tokens of type T_i to participant A_i (for all i). The needed funds are taken from the process balance and from the contributing participants.
- $\mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{\mathcal{E}_1}\,;?\rangle \| \cdots \| \mathtt{X}_n\langle \boldsymbol{\mathcal{E}_n}\,;?\rangle)$. This branch invokes the clauses $\mathtt{X}_i$ in parallel, with the actual internal parameters given by the *expressions* $\boldsymbol{\mathcal{E}_i}$. The external parameters $?$ represent values chosen by participants. The $\mathtt{call}$ operation is enabled when the funding preconditions of *every* $\mathtt{X}_i$ are satisfied. Like in the previous case, the needed funds are taken from the process balance, and from additional funds possibly sent by participants.

Branches can be decorated with time constraints and authorizations. Time constraints enable a branch only after a certain time has passed. They can be absolute ($\mathtt{after}\ t : D$), making the branch D enabled since a certain time t, or relative ($\mathtt{afterRel}\ \delta : D$), making D enabled after a delay δ since the previous contract step. Authorizations ($\mathsf{A} : D$) enable a branch only when A has provided their authorization. Note that multiple authorizations are possible (e.g., $\mathsf{A} : \mathsf{B} : D$). In this case, all the

involved participants must agree on the chosen branch. Furthermore, if the branch is a call, then they must also agree on the values of the external parameters. Effectively, all the external parameters are chosen by the authorizers.

To stipulate a contract, participants spawn an instance of a clause, providing the funds required by its funding precondition $\boldsymbol{v}\mathsf{T}$. Note that $\boldsymbol{v}\mathsf{T}$ can be a sequence $v_1\mathsf{T}_1 \cdots v_n\mathsf{T}_n$, meaning that v_i units of each token T_i are required. Doing so activates the clause, running its process, which can in turn spawn instances of other clauses via `call`, transferring the control to them. Spawning multiple instances of clauses is possible by exploiting the inherent parallelism of the UTXO model. We take advantage of this parallelism in the following example.

Exploiting parallelism. We specify a "Ponzi scheme" contract as follows:

$$\mathtt{P}(v;A) = \{v\mathsf{T}\}\ \mathtt{call}\ (\mathtt{S}\langle 2v, A;\rangle \| \mathtt{P}\langle 2v;?\rangle \| \mathtt{P}\langle 2v;?\rangle)$$
$$\mathtt{S}(v, A;) = \{v\mathsf{T}\}\ \mathtt{send}\ (v\mathsf{T} \to A)$$

The clause `P` takes an integer v as an internal parameter, and a participant A as an external parameter (denoting the owner). The funding precondition requires v units of T. The clause `P` calls `S` along with two copies of itself (each one doubling the internal parameter v). The clause `S` simply transfers some tokens according to its internal parameters v and A (note that they are before the ";"). When $\mathtt{P}\langle v;A\rangle$ is active, to continue the contract we need to satisfy the preconditions of all called clauses, which require $6v\mathsf{T}$ overall. Since the contract balance is $v\mathsf{T}$, participants must provide $5v\mathsf{T}$. In practice, the owner A will need to convince two participants B and C to provide $2.5v\mathsf{T}$ each, in exchange for setting themselves as owners in the newly spawned copies of `P`. When the `call` is performed, the former owner A receives $2v\mathsf{T}$. If B and C later manage to enrol two other participants in the scheme, they will receive $4v\mathsf{T}$, gaining $4v\mathsf{T} - 2.5v\mathsf{T} = 1.5v\mathsf{T}$. Note that if C does not find new participants, but B does, B can still continue its process, since each copy of `P` executes independently. We remark that B and C are *not* known when A is enrolled: this is why we need to use external parameters.

Upon compilation, parallel active contracts can be concurrently executed by UTXO blockchain nodes by exploiting their internal parallelism. This would not be possible with a traditional stateful account-based implementation, where a single contract would process all transactions.

2.2. The UTXO model

ILLUM contracts can be executed on a bare-bone UTXO blockchain, with no Turing-complete scripting language and no gas mechanism. Basically, a UTXO model similar to Bitcoin's is enough, with the addition of custom tokens and *covenants* [16]–[18].

A transaction T in this UTXO model is a tuple with the following fields, similarly to Bitcoin:

- out is a sequence of *transaction outputs*, i.e. triples of the form $(\mathsf{arg}, \mathsf{scr}, \mathsf{val})$, where arg is a sequence of values (which we use to encode the contract state), scr is the script specifying the spending condition, and val encodes the tokens held by the output ($\boldsymbol{v}\mathsf{T}$).
- in is a sequence of *transaction inputs*, referring to the transaction outputs which are going to be spent by T. An input (T', i) refers to the i-th output of a previous transaction T'.
- wit is a sequence of *witnesses* (sequences of values), passed as parameters to the scripts of input transactions. More precisely, if $\mathsf{in}(j) = (\mathsf{T}', i)$, then $\mathsf{wit}(j)$ is the witness passed to the script $\mathsf{T}'.\mathsf{out}(i).\mathsf{scr}$.

As a simple example, we display below a transaction T_1 containing a single output, which holds $1\mathsf{T}$ and can be redeemed by any transaction carrying a signature of A in its wit field (referred to by rtx.wit).

T_1
...

out(1): {arg : ⋯, scr : versig(pk_{A}, rtx.wit), // verify signature
val : $1\mathsf{T}$}

In order to redeem the tokens held in a transaction output $\mathsf{T}.\mathsf{out}(j)$, a transaction T'' has to satisfy the spending condition $\mathsf{T}.\mathsf{out}(j).\mathsf{scr}$. This script can access the fields of T and T'', perform basic arithmetic, logical and cryptographic operations (hashing, signature verification of the redeeming transaction), and enforce time constraints. Covenant operators allow the script in T to constrain the scripts in T''. The covenant $\mathsf{verscr}(\sigma, n)$ mandates the n-th output of T'' to have a script equal to σ. The covenant $\mathsf{verrec}(n)$ requires the n-th output of T'' to have the same script of $\mathsf{T}.\mathsf{out}(j)$, the one currently being checked.

As an example, consider the following transaction redeeming (the only output of) T_1:

T_2

in(1): $(\mathsf{T}_1, 1)$
wit(1): $sig_{\mathsf{A}}(\mathsf{T}_2)$
out(1): {arg : pk_{A},
scr : versig(ctxo.arg.1, rtx.wit) and // verify signature
rtxo(1).val = $1\mathsf{T}$ and // preserve value
verrec(1), // preserve script
val : $1\mathsf{T}$}

The transaction T_2 can redeem its input, since it carries a witness (a signature of A) that satisfies the script $\mathsf{T}_1.\mathsf{out}(1).\mathsf{scr}$. Furthermore the assets in T_2's output do not exceed those in its inputs. The spending condition of T_2 is given by the arg field (containing A's public key pk_{A}), and the script scr. The script is a conjunction of three conditions:

- the witness of the redeeming transaction must contain a transaction signature, to be verified against the public key stored in the 1st element of the arg sequence of the current transaction output (denoted by ctxo.arg.1). In T_2, this is just pk_{A}.
- the 1st output of the redeeming transaction must have $1\mathsf{T}$ value (here $\mathsf{rtxo}(i)$ is the i-th output of the *redeeming* transaction).
- the 1st script of the redeeming transaction must be equal to the current one. This is enforced using the covenant $\mathsf{verrec}(1)$.

This transaction actually implements a sort of Non-Fungible Token (NFT). To transfer the NFT to B, A spends T_2 with a redeeming transaction T_3, writing her signature in the wit field of T_3, and setting the arg field to B's public key pk_{B}. Note that the script and the balance are preserved along transactions thanks to the covenant.

```
Init(;) = {0} call Bid⟨0;?,?⟩

Bid(oldBid;newBid,Bidder)
 = {newBid T  if  newBid > oldBid and Bidder ≠ Null}
    call (Bid⟨newBid;?,?⟩ ‖ Pay⟨newBid,Bidder;⟩)
 + after 1000 : Owner : send (newBid T → Owner)

Pay(v,A;) = {vT} send (vT → A)
```

Figure 1: An auction contract in ILLUM.

2.3. The ILLUM compiler

One of our main contributions is a compiler from ILLUM contracts to UTXO transactions. Intuitively, we encode active clauses into transaction outputs, where the val field records the contract balance, scr enforces the contract logic, and arg records the contract state.

The arg field of a transaction output encoding an active clause will have one entry for each actual parameter, and the following additional entries: name to represent the clause name, branch to represent the index of the executed branch, and nonce to keep the behaviour faithful to the ILLUM semantics. The scr field of *all* transactions outputs resulting from the compilation of an ILLUM contract is the same, and it is preserved along chains of transactions by using the verrec covenant.

Here we illustrate the compilation of an auction contract focussing on the construction of the script.

An auction contract. The contract (Figure 1) consists of three clauses: `Init` that takes no parameters and initialises the auction with a starting bid of 0 tokens; `Bid` that allows a *Bidder* to raise the *oldBid* to a *newBid*; and `Pay` which transfers tokens to a participant. The contract flows as follows. After the initialisation, a participant can call the clause `Bid` to start the auction, setting themselves as the highest bidder. Then, we must execute one of the two branches of `Bid`. The first branch raises the bid, setting the values of *newBid* and *Bidder* through external parameters, and refunds the previous bidder. The second branch closes the auction, transferring the tokens to the hardcoded Owner. Since the first branch of `Bid` recursively calls the clause `Bid` itself, it can only be taken if the funding precondition is satisfied, which means that the new bid must be greater than the previous one. On the other hand, the second branch of `Bid` can only be executed by the owner after a deadline of 1000 time units. By choosing this branch the owner closes the auction, and receives the highest bid.

Compiling the contract, we obtain the following script:

$$\begin{aligned}\mathsf{scr}_{\mathsf{Auction}} \coloneqq\ & \mathsf{inidx} = 1 \text{ and} \\ & \text{if } (\mathsf{ctxo.name} = \texttt{Init}) \text{ then } \mathsf{scr}_{\mathsf{Init}} \text{ else} \\ & \text{if } (\mathsf{ctxo.name} = \texttt{Bid}) \text{ then } \mathsf{scr}_{\mathsf{Bid}} \text{ else} \\ & \text{if } (\mathsf{ctxo.name} = \texttt{Pay}) \text{ then } \mathsf{scr}_{\mathsf{Pay}} \text{ else } \mathit{false}\end{aligned}$$

The condition $\mathsf{inidx} = 1$ checks that this output is redeemed by an input at position 1. This is needed to thwart attacks where a transaction spends two contracts at once, effectively cancelling one of them. The rest of the script is a switch among the possible clauses, where ctxo.name denotes the name item of the arg field in the current transaction output. For brevity here we only illustrate the most interesting case, i.e. $\mathsf{scr}_{\mathsf{Bid}}$. Recall that the `Bid` process is a choice between two branches. Consequently, the associated script has the following form:

$$\begin{aligned}\mathsf{scr}_{\mathsf{Bid}} \coloneqq\ & \text{if BranchCond1 then } \mathsf{scr}_{\mathsf{Branch1}} \\ & \text{else if BranchCond2 then } \mathsf{scr}_{\mathsf{Branch2}} \\ & \text{else } \mathit{false}\end{aligned}$$

The first branch calls *two* clauses, i.e. `Bid` and `Pay`:

$$\texttt{call}\ (\texttt{Bid}\langle \mathit{newBid};?,?\rangle \parallel \texttt{Pay}\langle \mathit{newBid},\mathit{Bidder};\rangle)$$

Consistently, BranchCond1 checks that the redeeming transaction (rtx) has exactly *two* outputs: this is the goal of the condition $\mathsf{outlen}(\mathsf{rtx}) = 2$ below. Furthermore, BranchCond1 checks that the chosen branch is indeed the first one: this is done by requiring that both outputs in the redeeming transaction ($\mathsf{rtxo}(1)$ and $\mathsf{rtxo}(2)$) have the argument branch set to 1.

$$\begin{aligned}\mathsf{BranchCond1} \coloneqq\ & \mathsf{outlen}(\mathsf{rtx}) = 2 \text{ and } \mathsf{rtxo}(1).\mathsf{branch} = 1 \\ & \text{and } \mathsf{rtxo}(2).\mathsf{branch} = 1\end{aligned}$$

The second branch instead performs a `send` operation with exactly *one* recipient, so we check that the redeeming transaction has exactly *one* output, which has its branch argument set to 2.

$$\mathsf{BranchCond2} \coloneqq\ \mathsf{outlen}(\mathsf{rtx}) = 1 \text{ and } \mathsf{rtxo}(1).\mathsf{branch} = 2$$

The script $\mathsf{scr}_{\mathsf{Branch1}}$ verifies that the two outputs of the redeeming transaction encode, respectively, the clauses `Bid` and `Pay`. The part corresponding to `Bid` is:

$$\begin{aligned}& \mathsf{rtxo}(1).\mathsf{name} = \texttt{Bid} \text{ and} \\ & \mathsf{verrec}(1) \text{ and } |\mathsf{rtxo}(1).\mathsf{arg}| = 6 \text{ and} \\ & \mathsf{rtxo}(1).\mathsf{oldBid} = \mathsf{ctxo.newBid} \text{ and} \\ & \mathsf{rtxo}(1).\mathsf{val} = \mathsf{rtxo}(1).\mathsf{newBid}\ \mathrm{T} \text{ and} \\ & \mathsf{rtxo}(1).\mathsf{newBid} > \mathsf{rtxo}(1).\mathsf{oldBid} \text{ and} \\ & \mathsf{rtxo}(1).\mathsf{Bidder} \neq \mathsf{Null} \text{ and } \cdots\end{aligned}$$

The script performs the following checks on the redeeming transaction: (i) the clause name in the first output is indeed `Bid`; (ii) the script is preserved (via the verrec covenant); (iii) the number of arguments is correct (3 arguments for newBid, oldBid and Bidder, and 3 arguments for name, branch and nonce); (iv) the value of the oldBid of the redeeming transaction is set to the newBid of the current transaction, coherently with the passing of parameters in the `Bid` clause; (v) the amount of tokens of type T transferred to the redeeming transaction is the one specified in the funding precondition; (vi) the guard in the funding precondition is satisfied.

The part of the script corresponding to `Pay` is obtained in the same way:

$$\begin{aligned}\cdots\ & \mathsf{rtxo}(2).\mathsf{name} = \texttt{Pay} \text{ and} \\ & \mathsf{verrec}(2) \text{ and } |\mathsf{rtxo}(2).\mathsf{arg}| = 5 \text{ and} \\ & \mathsf{rtxo}(2).\mathsf{A} = \mathsf{ctxo.Bidder} \text{ and} \\ & \mathsf{rtxo}(2).\mathsf{v} = \mathsf{ctxo.newBid} \text{ and} \\ & \mathsf{rtxo}(2).\mathsf{val} = \mathsf{ctxo.newBid}\ \mathrm{T}\end{aligned}$$

The script $\mathsf{scr}_{\mathsf{Branch2}}$ encodes the `send` operation in Figure 1, enforcing the authorization of Owner and the absolute time constraint of 1000 time units:

$$\begin{aligned}&\mathsf{absAfter}\ 1000 : \mathsf{versig}(\mathsf{Owner}, \mathsf{rtx.wit}(1))\ \text{and}\\&\mathsf{verscr}(\mathsf{versig}(\mathsf{ctxo.owner}, \mathsf{rtx.wit}(1)), 1)\ \text{and}\\&|\mathsf{rtxo}(1).\mathsf{arg}| = 3\ \text{and}\\&\mathsf{rtxo}(1).\mathsf{owner} = \mathsf{Owner}\ \text{and}\\&\mathsf{rtxo}(1).\mathsf{val} = \mathsf{ctxo.newBid}\ \mathrm{T}\end{aligned}$$

The script performs the following checks on the redeeming transaction: (i) absAfter forces the redeeming transaction to be published at a time greater than 1000; (ii) the first versig checks the presence of Owner's signature in the witness of the redeeming transaction; (iii) the only output of the redeeming transaction has 3 arguments (owner, branch, and nonce), and a script that accepts any transaction signed by the owner (this is enforced by the verscr covenant). This effectively transfers the ownership of the tokens to Owner. The details of our compilation technique are in Section 4.

2.4. Security of the ILLUM compiler

Our main technical result is the security of the compiler establishing a strict correspondence between actions at the ILLUM level and those at the blockchain level. This ensures that any contract behaviour that is observable at the blockchain level is also observable in the semantics of ILLUM. In particular, any attack that may happen at the blockchain level can be detected by inspecting the symbolic semantics of ILLUM contracts. This is a fundamental step towards static analysis tools for the verification of security properties of contracts in the UTXO model.

Here we outline how this result is proved in Sections 5 and 6. We start by defining the adversary model, both at the symbolic level of ILLUM and at the computational level of the blockchain. The adversary is modelled as a PPTIME algorithm that schedules the actions chosen by participants, possibly interleaving them with adversarial actions. The bridge between the symbolic and the computational level is given through a *coherence relation*, which associates symbolic actions (e.g., a `call` action) with their computational counterparts (e.g., a transaction). The definition of this coherence relation is quite gruelling, as it must consider all possible actions, which amount to 20 cases: this complexity of course reflects on the proofs. As a first sanity check, we show in Lemma 1 that the coherence relation precisely characterises the exchange of assets: namely, the asset ownership is consistent between symbolic and computational executions whenever they are coherent. Our main security result (Theorem 2) guarantees that any computational execution is coherent with some symbolic execution, up-to a negligible error probability. Together with Lemma 1, it proves that computational exchanges of assets, including those mediated by contracts, are always mirrored at the symbolic level.

3. The ILLUM intermediate language

We now refine the description of ILLUM given in Section 2, by providing its syntax and semantics. Because of space constraints, we omit some technicalities, relying on examples and intuitions: see [21] for full details.

Syntax. We assume a set Part of participants, (ranged over by $\mathsf{A}, \mathsf{B}, \cdots$, and by a dummy participant Null). Contracts and deposits are denoted by the lowercase letters $x, y, \cdots$, while clauses will have names $\mathtt{X}, \mathtt{Y}, \cdots$. Clause parameters will be denoted by In_i and Ex_i, while the actual values substituted to those parameters will be denoted by $\overline{\mathrm{In}}_i$ and $\overline{\mathrm{Ex}}_i$. Arithmetic expressions (integer constants and parameters, basic operations, hashes) are denoted by $\mathcal{E}_i$, while participant expressions (constants and parameters) are denoted by $\mathcal{N}_i$. The value of parameters and expressions can also be key-value mappings. The domain of a mapping can be chosen to be either the integers or the participants. The codomain can be chosen similarly. If $\mathcal{M}$ is a mapping expression, we denote with $\mathcal{M}[\cdots]$ the access to one of its values, and with $\mathcal{M}[\cdots \to \cdots]$ the update of one of its associations. Sequences are denoted in bold, with $\boldsymbol{x} = x_1 \cdots x_n$.

We remark that the precise set of data types that can be used in parameters and expressions is not fundamental to the design of ILLUM. Indeed, our design can be easily adapted to different data types by suitably altering the syntax and semantics of expressions. To support compilation to the UTXO model, we simply require that the underlying blockchain scripting language supports the same data types and operations. We assume that data types include at least integers and participants, since their rôle is crucial to ILLUM constructs. Key-value mappings, instead, are not as crucial, and could be removed if not supported by the underlying blockchain, at the expense of reducing the usability of ILLUM. Throughout the paper, we mostly showcase examples that use only the fundamental data types (integers and participants). Notably, the ILLUM compiler handles all these contracts without requiring mappings to be supported by the compilation target. In Section 7 we will also exploit mappings to discuss more complex contracts.

Definition 1 (Clauses)**.** A clause is defined by an equation

$$\mathtt{X}(\boldsymbol{In}\,;\boldsymbol{Ex}) = \{\boldsymbol{\mathcal{E}}\mathrm{T}\ \ \mathtt{if}\ \ p\}\ \ C$$

where $\{\boldsymbol{\mathcal{E}}\mathrm{T}\ \ \mathtt{if}\ \ p\}$ is the *funding precondition*, and C is a *process*. The clause takes two sequences of parameters $\boldsymbol{In}$ and $\boldsymbol{Ex}$. Parameters can be of any type (integers, participants, or mappings). These types are always clear from the context, hence omitted. We require that all the parameter names in $\boldsymbol{\mathcal{E}}$, p, and C are present in $\boldsymbol{In}$, $\boldsymbol{Ex}$.

When X is invoked, the calling process provides the actual internal parameters, while the participants who are performing the call choose the actual external ones. The funding precondition $\{\boldsymbol{v}\mathrm{T}\ \ \mathtt{if}\ \ p\}$ encodes the requirements for the invocation of X. Namely, the sequence $\boldsymbol{v}\mathrm{T} = v_1\mathrm{T}_1 \cdots v_n\mathrm{T}_n$ asks the participants to transfer v_i tokens of type T_i to the process C (for all i). Moreover, p is a boolean condition on the parameters that must hold. We write $\{\boldsymbol{v}\mathrm{T}\}$ for $\{\boldsymbol{v}\mathrm{T}\ \ \mathtt{if}\ \ \mathit{true}\}$.

Definition 2 (Processes)**.** Processes have the following

syntax:

$C ::= \sum_{i\in I} D_i$	process
$D ::=$	branch
$\texttt{call}\ (\cdots \parallel \texttt{X}_i\langle \overline{\textbf{In}_i}\,;?\rangle \parallel \cdots)$	call clauses $\cdots \texttt{X}_i \cdots$
$\mid \texttt{send}\ (\cdots \parallel \mathcal{E}_i \mathrm{T}_i \to \mathcal{N}_i \parallel \cdots)$	transfer $\mathcal{E}_i \mathrm{T}_i$ to each $\mathcal{N}_i$
$\mid \mathcal{N} : D$	wait for $\mathcal{N}$ authorization
$\mid \texttt{after}\ \mathcal{E} : D$	wait until time $\mathcal{E}$
$\mid \texttt{afterRel}\ \mathcal{E} : D$	wait $\mathcal{E}$ after activation

where we assume that: (i) each clause name $\texttt{X}$ has a unique defining equation $\texttt{X}(\boldsymbol{In}\,;\boldsymbol{Ex}) = \{\boldsymbol{\mathcal{E}}\mathrm{T}\ \texttt{if}\ p\}\ C$; (ii) the sequence $\overline{\textbf{In}}$ of actual parameters passed to a called clause $\texttt{X}_i$ matches, in length and typing, the sequence $\boldsymbol{In}$ of formal (internal) parameters; (iii) the order of decorations is immaterial.

A clause $\texttt{X}(\cdots)$ together with two correctly typed sequences of actual parameters $\overline{\textbf{In}}$ and $\overline{\textbf{Ex}}$ is said to be an *instantiated clause*, and denoted by $\texttt{X}\langle\overline{\textbf{In}}\,;\overline{\textbf{Ex}}\rangle$.

Semantics. The execution of contracts is modelled as a transition relation between ***configurations***, that are abstract representations of the blockchain state. In a configuration, tokens can be stored in deposits and active contracts.

A ***deposit*** $\langle \mathsf{A}, v\mathrm{T}\rangle_x$ represents v units of token T owned by A. It is uniquely identified by the name x, and can only be spent upon A's authorization. A term $\langle C, \boldsymbol{v}\mathrm{T}\rangle^t_x$ is an ***active contract***, where x is the unique identifier, t is the activation time, and $\boldsymbol{v}\mathrm{T}$ is the balance, which can only be transferred according to the contract logic specified by C.

Besides the terms used to store tokens, in a configuration we also have advertisements and authorizations.

Advertisement terms are used by a participant to propose one of the following actions:

- The activation of a new contract. This is done by advertising $\Phi = \left[\texttt{X}\langle\overline{\textbf{In}}\,;\overline{\textbf{Ex}}\rangle\ ;\ \boldsymbol{z}\right]_h$, which specifies the instantiated clause $\texttt{X}\langle\overline{\textbf{In}}\,;\overline{\textbf{Ex}}\rangle$ and a nonempty list $\boldsymbol{z}$ of deposit names that will be spent to fund the contract. The index h is just a nonce used to differentiate between two otherwise identical advertisements.
- The continuation of an active contract. This is done by advertising $\Phi = \left[\bar{D}\ ;\ \boldsymbol{z}\ ;\ (x\,,j)\right]_h$. The list $\boldsymbol{z}$ specifies the deposits that will be spent for the continuation and added to the balance of x. The index h is again a nonce. The term $\bar{D}$ is an *advertised branch*, constructed by taking D, the j-th branch of x, and instantiating the question marks $?$ appearing in a $\texttt{call}$ with the actual values $\overline{\mathrm{Ex}}$.

Authorization terms are used by participants to enable the spending of deposits and to enable the execution of a contract branch decorated by $\mathsf{A} : \cdots$. Authorizations have the form $\mathsf{A}[\chi]$, where A is the authorizing participant, and χ denotes the authorized action. We see here two cases of authorization terms, relegating the others to Appendix A:

- $\mathsf{A}[z \rhd \Phi]$ authorises the spending of a deposit z owned by A that appears in the advertisement Φ;
- $\mathsf{A}[x \rhd \Phi]$ authorises the continuation of the j-th branch of a contract x, as advertised by $\Phi = \left[\mathsf{A} : \bar{D}\ ;\ \boldsymbol{z}\ ;\ (x\,,j)\right]_h$.

Definition 3 (Configurations)**.** A configuration Γ is a term $\tilde{\Gamma} \mid t$, where $t \in \mathbb{N}$ denotes the time, and the pre-configuration $\tilde{\Gamma}$ has the following syntax:

$\tilde{\Gamma} ::= \langle C, \boldsymbol{v}\mathrm{T}\rangle^t_x$	active contract
$\mid \langle \mathsf{A}, v\mathrm{T}\rangle_x$	deposit
$\mid \Phi$	advertisement
$\mid \mathsf{A}[\chi]$	authorization
$\mid (\tilde{\Gamma} \mid \tilde{\Gamma})$	parallel composition

We assume that: (i) the parallel composition is associative and commutative; (ii) all parallel terms are distinct; (iii) names are unique; (iv) all expressions occurring in active contracts are reduced to constants (integers or names).

An example. In Section 2 we have discussed the intuition behind the language semantics. Here we refine this intuition by precisely illustrating the evolution of the configuration during the execution of a simple contract. This example shows the semantics of the main language constructs, and the role of advertisements and authorizations terms.

$$\begin{aligned}\texttt{X}(a\,;b) &= \{b\mathrm{T}\ \texttt{if}\ b > a\}\ \mathit{Wait}(b)\\ \mathit{Wait}(b) &= \texttt{afterRel}\ 10 : \texttt{send}\ (b\mathrm{T} \to \mathsf{A})\\ &\quad + \mathsf{B} : \texttt{call}\ \texttt{X}\langle b\,;?\rangle\end{aligned}$$

The first branch allows A to withdraw the whole balance after 10 time units since the contract activation. The second branch allows B to temporarily prevent A from withdrawing: this requires B to restart the contract with an increased balance. We start from the initial configuration:

$$\Gamma_0 = \langle \mathsf{C}, 1\mathrm{T}\rangle_{z_1} \mid \langle \mathsf{B}, 2\mathrm{T}\rangle_{z_2} \mid \langle \mathsf{B}, 3\mathrm{T}\rangle_{z_3} \mid t$$

Participant C starts by advertising $\Phi_0 = [\texttt{X}\langle 0\,;1\rangle\ ;\ z_1]_h$, then authorizes the use of their deposit z_1 in the stipulation, and finally stipulates the contract, reaching configuration Γ_1:

$$\begin{aligned}\Gamma_0 &\to \Gamma_0 \mid \Phi_0 \ \to\ \Gamma_0 \mid \Phi_0 \mid \mathsf{C}[z_1 \rhd \Phi_0]\\ &\to \langle \mathit{Wait}(1), 1\mathrm{T}\rangle^t_x \mid \langle \mathsf{B}, 2\mathrm{T}\rangle_{z_2} \mid \langle \mathsf{B}, 3\mathrm{T}\rangle_{z_3} \mid t = \Gamma_1\end{aligned}$$

From Γ_1 there are multiple possible continuations. For instance, B can choose to execute the second branch of Wait. To do so, B first produces the advertisement $\Phi_1 = [\mathsf{B} : \texttt{call}\ \texttt{X}\langle 1\,;3\rangle\ ;\ z_2\ ;\ (x\,,2)]_h$. Then, B gives two authorizations: one to satisfy the decoration $\mathsf{B} : \cdots$ in the second branch of Wait, and another one to allow the spending of z_2. With these, the configuration can evolve as follows:

$$\begin{aligned}\Gamma_1 &\to \Gamma_1 \mid \Phi_1 \ \to\ \Gamma_1 \mid \Phi_1 \mid \mathsf{B}[x \rhd \Phi_1]\\ &\to \Gamma_1 \mid \Phi_1 \mid \mathsf{B}[x \rhd \Phi_1] \mid \mathsf{B}[z_2 \rhd \Phi_1]\\ &\to \langle \mathit{Wait}(3), 3\mathrm{T}\rangle^t_{x'} \mid \langle \mathsf{B}, 3\mathrm{T}\rangle_{z_3} \mid t = \Gamma_2\end{aligned}$$

B can again choose the second branch, this time spending z_3 to fund its execution. Otherwise, if B lets the time pass, A can advertise the continuation:

$$\Phi_2 = [\texttt{afterRel}\ 10 : \texttt{send}\ (3\mathrm{T} \to \mathsf{A})\ ;\ ;\ (x'\,,1)]_h$$

and then withdraw the contract balance:

$$\begin{aligned}\Gamma_2 &\to \langle \mathit{Wait}(3), 3\mathrm{T}\rangle^t_{x'} \mid \langle \mathsf{B}, 3\mathrm{T}\rangle_{z_3} \mid t + 10 = \Gamma_3\\ &\to \Gamma_3 \mid \Phi_2 \ \to\ \langle \mathsf{A}, 3\mathrm{T}\rangle_{z_4} \mid \langle \mathsf{B}, 3\mathrm{T}\rangle_{z_3} \mid t + 10\end{aligned}$$

The semantics of ILLUM has a set of rules for reducing contracts, and another set for deposits (see Appendix A).

Turing-completeness. ILLUM is Turing-complete: indeed, we can simulate in ILLUM any counter machine [22], a well-known Turing-complete computational model. The proof is similar to that in [23]: we simulate any counter machine by storing each counter in the arguments of recursive clauses. Incrementing and decrementing the counters is simply done by specifying the new values of the arguments inside the `call`. Conditional jumps are simulated as choices, also exploiting clause preconditions. This construction does not exploit key-value mappings: it is only based on the assumption that integers are unbounded, as usual. Notice that, despite its Turing-completeness, ILLUM can be compiled down to a "poor" UTXO blockchain, i.e. one with non-Turing-complete scripts. This is accomplished by spreading the execution of a compiled contract across multiple transactions, each with its own loop-free script. Note that our key-value mappings just feature operators to lookup a single key and to update a single association: in this way, even with maps, UTXO scripts can be run in nearly constant-time. This makes the gas mechanism unnecessary.

4. Compiling ILLUM to UTXO scripts

The compilation target of ILLUM is a UTXO blockchain that is close to Bitcoin, with minimal extensions in the structure of transactions and in the scripting language to overcome its expressiveness limitations.

Scripting language. We consider a scripting language that extends Bitcoin Script with covenants, borrowing from [16] (see [21] for its syntax and semantics). Here we recap some operators that are used by our compiler. First, $\mathsf{ctxo.f}$ denotes the field f of the current transaction output that is being spent. Similarly, $\mathsf{rtxo}(e).\mathsf{f}$ denotes the field f of the e-th output of the redeeming transaction. Then, we have the covenants: $\mathsf{verscr}(\mathsf{scr}, n)$ checks that the n-th output script of the redeeming transaction is equal to scr, while $\mathsf{verrec}(n)$ checks that the n-th output script of the redeeming transaction is equal to the one of the output being spent. The operators inidx and rtxw denote, respectively, the position of the redeeming input among the ones of the redeeming transaction, and the witness associated to it. To improve readability, we will use names instead of indices when referring to arguments in the arg sequence (e.g., we write $\mathsf{ctxo.owner}$ for $\mathsf{ctxo.arg}.1$).

While constructing a contract script we will need to replace the parameters In_i and Ex_i appearing in an expression $\mathcal{E}$ with the respective arguments $\mathsf{ctxo.In_i}$ and $\mathsf{ctxo.Ex_i}$. To simplify the notation, we denote this substitution with $\mathsf{ctxo}.\mathcal{E}$. For instance, if the contract contains a term $\texttt{after}\, t : \cdots$ with $t = In_1 + Ex_1 + 3$, we will write $\mathsf{ctxo}.t$ instead of $\mathsf{ctxo.In_1} + \mathsf{ctxo.Ex_1} + 3$. Similarly, whenever an expression uses the arguments of a redeeming transaction's output, we denote it as $\mathsf{rtxo}(j).\mathcal{E}$.

Representing deposits. We represent a deposit $\langle \mathsf{A}, v\mathrm{T} \rangle$ in an ILLUM configuration as a transaction output with value $v\mathrm{T}$, argument owner set to A, and the script:

$$\mathsf{versig}(\mathsf{ctxo.owner}, \mathsf{rtxw}.1)$$

allowing A to spend the funds by providing her signature.

How the compiler works. Representing an active contract $\langle C, \boldsymbol{v}\mathrm{T} \rangle_x^t$ at the blockchain level is more complex: we need to consider the clause $\mathtt{X}\langle \overline{\mathbf{In}}; \overline{\mathbf{Ex}} \rangle$ from which it originated. The output representing x has a value $\boldsymbol{v}\mathrm{T}$, and its arguments are the following: name for the clause name $\mathtt{X}$, for the actual parameters $\mathsf{In_i}$ and $\mathsf{Ex_i}$, and two technical arguments nonce and branch. The output script of a contract is preserved along executions: we detail its construction in the next paragraphs, refining and generalising the intuitions given in Section 2 (the full technical details are in [21]).

Let $\mathtt{X}_0$ be the initial clause of a contract, and let $\mathtt{X}_1 \cdots \mathtt{X}_n$ be the clauses that can be reached by recursively following every `call` operation that appears in $\mathtt{X}_0$'s definition. We assume that $\mathtt{X}_i$ defines the process C_i. We generate a script for the overall contract as follows. The script requires that the output must be redeemed from an input in the first position. Then, it performs a switch on the name argument to see which clause is currently encoded in the transaction output, and choose accordingly which script is going to be executed:

$$\begin{aligned} \mathsf{scr} \coloneqq\ & \mathsf{inidx} = 1 \text{ and} \\ & \text{if } (\mathsf{ctxo.name} = \mathtt{X}_0) \text{ then } \mathsf{scr}_{X_0} \text{ else} \\ & \vdots \\ & \text{if } (\mathsf{ctxo.name} = \mathtt{X}_n) \text{ then } \mathsf{scr}_{X_n} \text{ else } \mathit{false} \end{aligned}$$

To construct the script associated to a clause $\mathtt{X}$ in $\mathtt{X}_0 \cdots \mathtt{X}_n$, we inspect its process $C = D_1 + \cdots + D_m$, and associate an integer n_j with each D_j as follows: if D_j ends in a `call`, then n_j is equal to the number of called clauses; otherwise, if D_j ends in a `send`, then n_j is equal to the number of participants receiving the funds. This n_j will be the number of outputs of a transaction that redeems the j-th branch of C. This transaction must also specify the value j in the branch argument of each of its outputs. To check these conditions, we use:

$$\begin{aligned} \mathsf{B}_j \coloneqq\ & (\mathsf{outlen}(\mathsf{rtx}) = n_j \text{ and } \mathsf{rtxo}(1).\mathsf{branch} = j \text{ and} \\ & \cdots \text{ and } \mathsf{rtxo}(n_j).\mathsf{branch} = j) \end{aligned}$$

Then, we handle all the branches, with a conditional

$$\begin{aligned} \mathsf{scr}_X \coloneqq\ & \text{if } \mathsf{B}_1 \text{ then } \mathsf{scr}_{D_1} \text{ else} \\ & \vdots \qquad \vdots \qquad \vdots \\ & \text{if } \mathsf{B}_m \text{ then } \mathsf{scr}_{D_m} \text{ else } \mathit{false} \end{aligned}$$

Each branch D in $D_1 \cdots D_m$ is a sequence of decorations ended by a `call` or `send`. To construct scr_D, we first focus on the decorations. If there is an authorization decoration, then the witnesses of the redeeming transaction requires a signature by the authorizing participant:

$$\begin{aligned} \mathsf{scr}_{\mathsf{A}_1 : \cdots : \mathsf{A}_k : D'} \coloneqq\ & \mathsf{versig}(\mathsf{ctxo.A}_1, \mathsf{rtxw}.1) \text{ and } \cdots \\ & \text{and } \mathsf{versig}(\mathsf{ctxo.A}_k, \mathsf{rtxw}.k) \text{ and } \mathsf{scr}_{D'} \end{aligned}$$

where D' does not contain any authorization decoration. The "after" decorations are handled by the corresponding script operators absAfter/relAfter for absolute/relative timelocks:

$$\begin{aligned} \mathsf{scr}_{\texttt{after}\, t\, :\, D''} &\coloneqq \mathsf{absAfter}\ \mathsf{ctxo}.t : \mathsf{scr}_{D''} \\ \mathsf{scr}_{\mathsf{afterRel}\, \delta\, :\, D''} &\coloneqq \mathsf{relAfter}\ \mathsf{ctxo}.\delta : \mathsf{scr}_{D''} \end{aligned}$$

Finally, we describe the terminal parts of the script, i.e. `send` and `call`. First, we consider the case:

$$D'' = \texttt{send}\ (v_1\mathrm{T}_1 \to \mathsf{A}_1 \parallel \cdots \parallel v_n\mathrm{T}_n \to \mathsf{A}_n)$$

Here, we want each output of the redeeming transaction to encode a deposit of value $v_k\mathrm{T}_k$ owned by A_k. We use the operator verscr to force the redeeming transaction to have the correct script, and $|\cdot|$ to check that it has exactly 3 arguments (corresponding to nonce, branch, and owner). The script also checks the output values and the owners:

$$\mathsf{scr}_{D''} \coloneqq \left.\begin{array}{l} |\mathsf{rtxo}(i).\mathsf{arg}| = 3 \text{ and} \\ \mathsf{verscr}(\mathsf{versig}(\mathsf{ctxo}.\mathsf{owner}, \mathsf{rtxw}.1), i) \text{ and} \\ \mathsf{rtxo}(i).\mathsf{owner} = \mathsf{ctxo}.\mathsf{A}_i \text{ and} \\ \mathsf{rtxo}(i).\mathsf{val} = \mathsf{ctxo}.v_i\mathrm{T}_i \text{ and } \cdots \end{array}\right\} i \in 1..n$$

The last case is that for a `call`:

$$D'' = \texttt{call}\ (\mathtt{Y}_1\langle\overline{\mathbf{In_1}}\,;?\rangle \parallel \cdots \parallel \mathtt{Y}_n\langle\overline{\mathbf{In_n}}\,;?\rangle)$$

Let $\mathtt{Y}_i(\boldsymbol{In_i}\,;\boldsymbol{Ex_i}) = \{\boldsymbol{\mathcal{E}_i}\mathrm{T}_i \ \texttt{if}\ p_i\}\ C_i$, where $|\boldsymbol{In_i}| = k_i$ and $|\boldsymbol{Ex_i}| = h_i$. The script $\mathsf{scr}_{\mathtt{call}i}$ requires that the i-th output of the redeeming transaction encodes the contract specified by $\mathtt{Y}_i\langle\overline{\mathbf{In_i}}\,;\overline{\mathbf{Ex_i}}\rangle$, for some choice of the parameters $\overline{\mathbf{Ex_i}}$. We use verrec to preserve the contract script, and then check that the output has the correct number of arguments. We also require that the name is $\mathtt{Y}_i$, and that the arguments In_{ij} match the actual parameters.

Finally, we check the funding precondition:

$$\begin{array}{rl} \mathsf{scr}_{\mathtt{call}i} \coloneqq & \mathsf{verrec}(i) \text{ and } |\mathsf{rtxo}(i)| = 3 + k_i + h_i \text{ and} \\ & \mathsf{rtxo}(i).\mathsf{name} = \mathtt{Y}_i \text{ and} \\ & \mathsf{rtxo}(i).In_{\mathsf{i}1} = \mathsf{ctxo}.\overline{\mathsf{In}}_{i\,1} \text{ and } \cdots \text{ and} \\ & \mathsf{rtxo}(i).In_{\mathsf{i}k_i} = \mathsf{ctxo}.\overline{\mathsf{In}}_{i\,k_i} \text{ and} \\ & \mathsf{rtxo}(i).\mathsf{val} = \mathsf{rtxo}(i).\boldsymbol{\mathcal{E}_i}\ \mathrm{T}_i \text{ and} \\ & \mathsf{rtxo}(i).p_i \end{array}$$

This must be done for all $\mathtt{Y}_1 \cdots \mathtt{Y}_n$, obtaining:

$$\mathsf{scr}_{D''} \coloneqq \mathsf{scr}_{\mathtt{call}1} \text{ and } \cdots \text{ and } \mathsf{scr}_{\mathtt{call}n}$$

Executing a compiled contract. The ILLUM compiler translates an ILLUM contract into a script. In this way, the compilation process creates a correspondence between active contracts and transaction outputs on the blockchain. We will formalise this *coherence relation* later when establishing the security of the compiler. For now, we just note that each execution step of an active ILLUM contract corresponds, in the blockchain, to a new transaction redeeming the previous output.

A hint about the correctness of the compiler. The script produced by the compiler imposes very stringent conditions on the redeeming transaction T. In particular, its outputs are almost completely determined by the compiled script: the number of outputs and their assets are fixed; their script is determined either by verrec (in the `call` branches) or by verscr (in the `send` branches); the number of arguments is fixed, and the value of most of the arguments (which encode the contract state) is determined by the script. The only "free" fields in T are the arguments representing the external contract parameters, which are only subject to respect the funding precondition. This mirrors the ILLUM semantics, where participants can choose the actual external parameters at runtime. This "rigidity" is important in establishing the correctness of the ILLUM compiler: if a UTXO encodes an active contract $\langle C, \boldsymbol{v}\mathrm{T}\rangle^t_x$, then any transaction that redeems it must behave in "agreement" with one of the branches of C. The full details of the proof of correctness are presented in [21].

5. Adversary model

The semantics of ILLUM describes *all* actions that can be performed on contracts and deposits. For this reason the set of reachable configurations is very broad. In particular, it always contains the configuration obtained by donating all the deposits to a single participant. However, in a realistic scenario, this configuration would not be reached because participants would have no interest in authorizing the donations. To avoid considering these unrealistic executions, we need to restrict the semantics according to the behaviour of participants, which can decide whether to authorize or not any given action. To this aim, we introduce *strategies*, which are algorithms that model the participants' behaviour, computing the actions chosen by a participant at each execution step. We assume a subset Hon of participants for whom the strategies are known. The strategies of these *honest* participants are instrumental in defining the adversary model. Namely, we see the adversary Adv as an entity that controls the scheduling of the actions chosen by honest participants, and possibly inserts their own actions. This is consistent with adversarial miners/validators in blockchains, who can read user transactions in the mempool, and produce blocks containing some of these transactions, suitably reordered and possibly interleaved with their own transactions. Intuitively, we model the adversary as a strategy that controls all participants outside of Hon, can observe the actions outputted by the strategies of honest participants, and can decide to perform one of these actions or one of their own.

Symbolic runs. A symbolic run R^s is a sequence of configurations Γ_i connected by semantic actions α_i. The first configuration in the sequence only contains deposits, and has time $t_0 = 0$. We denote with Γ_{R^s} the last configuration of R^s. A run is written as $R^s = \Gamma_0 \xrightarrow{\alpha_0} \Gamma_1 \xrightarrow{\alpha_1} \cdots$.

Symbolic strategy of honest participants. Each honest participant A has a strategy Σ^s_{A}, i.e. a PPTIME algorithm that takes as input the symbolic run R^s and outputs the set of "choices" of A, i.e. the ILLUM actions that A wants to perform. The strategy is subject to well-formedness constraints: (i) the actions in $\Sigma^s_{\mathsf{A}}(R^s)$ must be enabled by the semantics in Γ_{R^s}; (ii) each authorization action in $\Sigma^s_{\mathsf{A}}(R^s)$ must be of the form $\mathsf{A}[\cdots \triangleright \cdots]$, forbidding A to impersonate another participant.

Symbolic adversarial strategy. Dishonest participants are controlled by the adversary Adv, who is also in charge of scheduling updates to the run. Their strategy Σ^s_{Adv} is a PPTIME algorithm that takes as inputs the run R^s and the sets of choices given by the honest participants' strategies. The output of Σ^s_{Adv} is a single ILLUM action λ^s that will be used to update the run. Adv's strategy is subject to the following constraints: (i) λ^s must be enabled in Γ_{R^s};

(ii) if λ^s is an authorization action by a honest A, then it must have been chosen by A; (iii) if λ^s is a delay, then it must have been chosen by all honest participants. The second condition prevents Adv from forging signatures, while the third condition ensures that Adv cannot prevent honest participants from meeting deadlines.

Symbolic conformance. Since a strategy Σ^s_{A} is probabilistic, we implicitly assume that it takes as input an infinite sequence of random bits r_{A}. Consider now a set of strategies $\boldsymbol{\Sigma}^s$ including those of honest participants, Σ^s_{Adv}, and a random source r from which the sequences r_{A} are derived. We can uniquely determine a run R^s by performing the actions outputted by Σ^s_{Adv}. Such a run is said to *conform* to $(\boldsymbol{\Sigma}^s, r)$.

Computational runs. Above, we have defined an adversarial model at the symbolic level. We model adversaries at the computational level in a similar way, replacing symbolic actions with computational ones. A computational run R^c is a sequence of actions λ^c in one of these forms:

T	appending transaction T to the blockchain
δ	performing a delay
$\mathsf{A} \to * : m$	broadcasting of message m from A

The first action in the run R^c is an initial transaction that distributes tokens to participants, and it is followed by the broadcast of each participant's public keys.

Computational strategies of honest participants. Each honest A is associated with a computational strategy, i.e. a PPTIME algorithm Σ^c_{A} that takes as input a computational run and outputs the set of choices of A. If $\Sigma^c_{\mathsf{A}}(R^c)$ includes an action $\lambda^c = \mathsf{T}$, then T must be *consistent* with R^c, essentially meaning that T is a valid transaction in the blockchain state reached after the run R^c.

Computational adversarial strategy. Like in the symbolic case, the adversary is given scheduling power. The strategy Σ^c_{Adv} takes as input the run R^c and the actions chosen by honest participants, and outputs a single action that will be used to update R^c. As for honest participants, the adversary cannot output invalid transactions. Like in the symbolic case, Σ^c_{Adv} is only allowed to output a delay if it has been chosen by all honest participants. We allow the adversary to impersonate any honest participants A. However, since Adv does not know the random source r_{A}, and Σ^c_{Adv} is PPTIME, Adv will be, with overwhelming probability, unable to forge A's signatures.

Computational conformance. Like in the symbolic case, a set of strategies $\boldsymbol{\Sigma}^c$ and a random source r can be used to uniquely determine a computational run R^c, that is said to conform to the pair $(\boldsymbol{\Sigma}^c, r)$.

6. Security of the ILLUM compiler

Symbolic and computational runs describe the evolution of contracts at two different level of abstraction: in Section 4 we have shown how transaction outputs encode ILLUM deposits and contracts. Formally this correspondence between runs is modelled as a relation, which we call *coherence*. Intuitively, coherence holds when the symbolic steps in R^s and the computational steps in R^c have the same effects on contracts and deposits.

Coherence. The coherence relation $R^s \sim_{txout} R^c$ is parameterized by a map $txout$ that relates the symbolic names of deposits and contracts to transaction outputs. Coherence is defined inductively, by exhaustively listing the possible actions of R^s. Here we present the most important cases, relegating the full definition to [21].

- Advertising Φ in R^s is matched by the broadcast of a message m in R^c. The message m encodes a transaction T_Φ representing the action advertised by Φ. In particular, the script of T_Φ's outputs must be the one produced by the compiler. Note that T_Φ is not yet appended to the blockchain, but only broadcast.
- Sending an authorization $\mathsf{A}[\cdots \rhd \Phi]$ in R^s is matched by sending a message m in R^c, containing a corresponding signature from A on T_Φ.
- Initiating a contract in R^c consumes the respective advertisement Φ and the required authorizations and deposits to insert a term $\langle C, \boldsymbol{v}\mathsf{T}\rangle^t_x$ in the configuration. This is matched in R^c by appending T_Φ to the blockchain. This uses the signatures that were broadcast together with the symbolic authorizations. Moreover, the map $txout$ is updated so that the new symbolic name x is mapped to T_Φ's output.
- Continuing a contract in R^c is similar to the contract initiation described above, except that it may produce multiple deposits or contracts instead of a single one. Again, it is matched in R^c by T_Φ, where Φ is the continuation advertisement, and $txout$ is updated to map the new names to each output of T_Φ.

The full definition also deals with deposit operations, delays, and transactions that spend inputs that are outside of the image of $txout$. The coherence relation is instrumental to establish correspondence results between the two models. Notably, Lemma 1 shows that the coherence relation precisely characterizes the exchange of assets.

Lemma 1. *If $R^s \sim_{txout} R^c$ and $\langle \mathsf{A}, v\mathsf{T}\rangle_x$ is a deposit appearing in the last configuration Γ_{R^s}, then $txout(x)$ is an unspent output in R^c and encodes the deposit x (i.e. it has the structure presented in Section 4). Notably, this means that the token balance is preserved by $txout$.*

This coherence result is lifted to an analogous lemma for contracts. We also establish the injectivity of the map $txout$, which ensures that no two distinct symbolic deposits or contracts are represented by the same transaction output. This means that the whole volume of assets is preserved by the map. Moreover, the coherence relation can be used as a guide to algorithmically translate the symbolic strategy of an honest participant into an equivalent computational strategy.

From symbolic to computational strategies. Here we present the map $\aleph$ that translate strategies. Given the symbolic strategy Σ^s_{A}, the computational strategy $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$ will do the following: first parse R^c to create a corresponding symbolic run R^s (using the coherence relation), then run $\Sigma^s_{\mathsf{A}}(R^s)$ producing a set of symbolic labels Λ^s, and lastly use the coherence again to transform

each symbolic label into the corresponding computational label, which will be the output of the strategy.

Security of the compiler. Theorem 2 gives us a way to construct a symbolic run R^s that is coherent to a given computational run R^c and conform to a set of given honest symbolic strategies $\mathbf{\Sigma}^s$. This is done under the assumption that R^c conforms to the honest computational strategies obtained by translating $\mathbf{\Sigma}^s$. By contrast, we make no assumption on the computational adversarial strategy used to construct R^c. Together with Lemma 1, computational exchanges of assets, including those mediated by contracts, are mirrored at the symbolic level.

Theorem 2 (Security of the compiler)**.** *Let $\mathbf{\Sigma}^s$ be a set of symbolic strategies for honest participants, let $\mathbf{\Sigma}^c = \aleph(\mathbf{\Sigma}^s)$, and let Σ^c_{Adv} be a computational adversarial strategy. If R^c is a run with polynomial length conforming to $\mathbf{\Sigma}^c \cup \{\Sigma^c_{\mathsf{Adv}}\}$, then there exist, with overwhelming probability, a symbolic run R^s and an adversarial strategy Σ^s_{Adv} such that (i) R^s is coherent with R^c, and (ii) R^s conforms to $\mathbf{\Sigma}^s \cup \{\Sigma^s_{\mathsf{Adv}}\}$.*

Proof (sketch). We match step-by-step the computational moves with the symbolic moves according to the coherence relation. In particular, looking at possible transactions, we have two main cases:

- A deposit operation (e.g. donating a deposit). This requires participant signatures. If in the symbolic run, there are the corresponding authorizations, then this operation has an immediate symbolic counterpart. Otherwise, the computational signatures have been forged, which happens with negligible probability.
- A contract operation (e.g., a call to a new clause). This can be done only with a transaction that satisfies the contract script. Since the script closely matches the symbolic semantics, we can construct the corresponding symbolic move (again, the only case where this is not possible is that of a signature forgery).

The full proof in [21] considers all the possible computational moves (e.g., outputting a message, waiting), and relates them to a specific symbolic action that maintains the coherence relation. Technically, this requires examining all the *twenty* cases in the definition of coherence and proving that whenever none of them applies, the adversary must have managed to forge signatures. □

The above security result can be seen in terms of Robust Trace Property Preservation (RTP) [19], [20]. RTP can be equivalently [20] stated in this form:

$$\forall \mathsf{P}.\forall \mathbf{C_T}.\forall t.\ \mathbf{C_T}[\mathsf{compile}(\mathsf{P})] \rightsquigarrow t \implies \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

This can be read in our setting as "whenever a ILLUM contract P is compiled and run in a computational adversarial context $\mathbf{C_T}$, producing an execution trace t, then there exists a symbolic adversarial context $\mathsf{C_S}$ where the original contract P produces the same trace t".

The above property can not be proved as-is in our setting, for a number of reasons. First, computational and symbolic traces have different nature, so we can not claim to have the same trace in both worlds – we instead claim that we have two traces R^c, R^s which are related by the coherence relation. Furthermore, computational adversaries always have a negligible probability to break the underlying cryptography, so RTP can only hold with overwhelming probability and for traces having polynomial length. The statement of Theorem 2 accounts for these peculiarities. Finally, in our formulation, the adversarial contexts are interpreted as (symbolic/computational) adversarial strategies.

7. From high-level languages to ILLUM

Although ILLUM provides an abstraction layer over the UTXO transaction model, its clause-based nature may make it unwieldy for developers familiar with the procedural style, which is currently mainstream in the smart contracts community thanks to languages like Solidity. We show in this section that it is possible to reconcile the UTXO model with the familiar high-level imperative procedural style. More specifically, we consider an expressive fragment of Solidity, and we show how to compile it down to ILLUM. We evaluate our approach by developing a prototype compiler and interpreter for the high-level language (∼2000 LoC of OCaml code), and by applying it to a benchmark of common smart contracts, including complex DeFi protocols like Automated Market Makers and Lending Pools. Overall, one can benefit from the formal security guarantees of ILLUM, while sticking to a familiar development process.

The HELLUM contract language. As a high-level language for contracts in the UTXO model, we consider a fragment of Solidity, a widespread smart contract language that has been popularized by Ethereum. To make the compilation into UTXO possible, we get rid of a couple of problematic features, i.e. loops and external contract calls. To compensate for the absence of external calls, which are the basis to implement custom tokens in Solidity, our language supports custom tokens natively.

The resulting High-Level Language for the UTXO Model, hereafter dubbed HELLUM, is exemplified in Figure 2 through a crowdfunding contract. The contract collects funds from donors until a `deadline`, then it transfers them to the `owner` only if the donations reach a given `target` amount. If the target is not met, then every donor is entitled to take back their donations. The constructor sets the contract parameters. The `next` modifier constrains which functions can be called next. The `deposit` function receives donations, and updates the map `funds` accordingly. The modifier `input(x:T)` requires the caller to pay an amount `x` of tokens `T` upon a call. The `require` command sets the minimum donation to 10 token units. The `finalize` function can only be called after the deadline is reached. If the collected funds (i.e. the contract balance) exceed the target, then they are transferred to the owner: otherwise the funds are kept in the contract. Executing `finalize` enables the `withdraw` function, through which donors can take back their donations if the target has not been reached (note that if the target was met, then `withdraw` transfers no funds). The modifier `auth(a)` requires that any withdraw to `a` must be authorized by the user controlling that address (i.e., the one who knows `a`'s private key).

We argue that this variant of Solidity is still practical for a wide range of applications (see Table 1). Regarding loops, we note that in general they are discouraged

```
contract Crowdfund {
  mapping (address => uint) funds;
  uint deadline;
  uint target;
  address owner;

  constructor(uint d, uint t, address o) {
    owner = o;
    deadline = d;
    target = t;
  } next(deposit,finalize)

  function deposit(uint x, address a) input(x:T) {
    require(x>=10);
    funds[a] += x;
  } next(deposit,finalize)

  function finalize() after(deadline) {
    if (balance(T) >= target)
      owner.transfer(balance(T):T);
  } next(withdraw)

  function withdraw(address a) auth(a) {
    a.transfer(funds[a]:T);
    funds[a] = 0;
  } next(withdraw)
}
```

Figure 2: A crowdfund contract in HELLUM.

```
function f(...) {
  require expr0; // this is the only require in f
  // chain of conditional statements
  if (cond1) {
    // sequence of token transfers
    a1_1.transfer(amt1_1:T1_1);
    ...
    a1_n1.transfer(amt1_n1:T1_n1);
    // single simultaneous assignment
    x_1,...,x_m = expr1_1,...,expr1_m;
  } else if (cond2) {
    a2_1.transfer(amt2_1:T2_1);
    ...
    a2_n2.transfer(amt2_n2:T2_n1);
    x_1,...,x_m = expr2_1,...,expr2_m;
  }
  ...
  else {
    ... // same structure as previous blocks
  }
}
```

Figure 3: Normal form of HELLUM functions.

even in Solidity, since they may vehicle gas exhaustion attacks [24] where an adversary causes an iteration to exceed the block gas limit, thereby making the users pay the gas fees for failed transactions, and, possibly, making the contract stuck [25]. Despite this limitation, our language features unbounded data structures, in the form of key/value mappings. Iterative behaviours can be obtained by shifting the duty of performing iterations to users, by requiring them to perform repeated calls to contract functions (see e.g. the withdrawal of funds in the crowdfunding contract). Regarding external calls, while in Solidity they are the basis for any interaction between a contract and the environment (including pure transfers of assets), in the UTXO model they are unnatural, since transaction validation must only involve the scripts referred to in the transaction inputs. Cardano, the main smart contract platform based on the UTXO model, does not support external calls. In our high-level language we use a special primitive `transfer` to exchange tokens, and we restrict calls to internal pure functions.

```
contract Test {
  uint x;

  function f(uint y, address a) {
    x = balance(T) - y;
    if (x > 10) {
      a.transfer(x:T);
      require balance(T) > 20;
      x += 1;
    }
  } next(f,g)

  function g() { }
}
```

Figure 4: A simple contract in HELLUM.

```
contract Test_NF {
 uint x;

 function f(uint y,address a) {
  require ((balance_pre(T)-y>10) && y>20) ||
           (balance_pre(T)-y<=10);
  if (balance(T)-y>10) {
   a.transfer(balance_pre(T)-y:T);
   x,bal_T_fin = (balance_pre(T)-y)+1,
    balance_pre(T)-(balance_pre(T)-y);
  }
  else {
   x,bal_T_fin = balance_pre(T)-y,balance_pre(T);
  }
 } next(f,g)

 function g() {
   x,bal_T_fin = x,balance_pre(T);
 }
}
```

Figure 5: Normal form of the `Test` contract.

Compiling HELLUM to ILLUM. HELLUM contracts can be automatically compiled to ILLUM. Here we summarize the translation process (see Appendix I for more details, and https://github.com/bitbart/illum-lang/for the implementation). We use the `Test` contract in Figure 4 as a working example to illustrate the compilation process.

First, we process each function in the HELLUM contract, passing it through code transformations which bring it to the normal form displayed in Figure 3. More specifically, a function is in normal form when:

- expressions do not contain internal calls to pure functions (these calls are macro-expanded);
- the function starts with a single `require` statement, which is the only one appearing in the function body;
- after the `require`, the rest of the function body is a chain of conditional statements;
- each conditional block starts with a sequence of `transfer` statements, followed by a single simultaneous assignment of all of the contract variables. This assignment also defines auxiliary variables representing the new contract balance after the transfers.

For example, the normal form obtained for the `Test` contract is displayed in Figure 5. In the transformed contract, we use the expression `balance_pre(T)` to denote the contract balance of token `T` *before* the function call, and the auxiliary variable `bal_T_fin` to denote the balance of `T` *after* the call. When in normal form, functions are amenable to be translated into ILLUM clauses, since the simultaneous assignments effectively specify the new contract state as a function of the old state.

The HELLUM compiler transforms each function `f`

Contract	HELLUM		ILLUM	
	LoC	B	LoC	B
Crowdfund	29	949	64	2150
Auction	31	772	52	1577
Payment splitter	37	1030	77	3183
Vault	39	984	90	4070
Automated Market Maker	40	1213	88	3642
Voting	42	1296	91	4519
Vesting wallet	44	1194	69	3360
Escrow	45	1359	99	3602
King of the hill	50	2062	69	4509
Blind auction	57	1742	86	5619
Lending pool	75	2062	132	6581
Lottery	78	2297	136	6401

TABLE 1: Benchmark of smart contracts in HELLUM, displaying lines of code (LoC) and size in bytes (B).

into two ILLUM clauses, called `f_run` and `f_next`. The clause `f_run` is used to take the parameters of `f` and run the function body. It has one branch for each conditional branch of `f`: these branches are enabled by the same conditional guards, and perform the payments alongside with calling `f_next` with the updated state passes as parameter. The funding precondition of `f_run` is computed taking into account the `input` modifiers, as well as the expression enclosed in the `require` statement.

The clause `f_next` allows the execution to continue by calling one of the contract functions, as constrained by the `next` modifier in the HELLUM function `f`. To this purpose, `f_next` has one branch for each of the possible continuation functions. The branches of `f_next` use the ILLUM decorators to implement the behaviour of the `auth` and `after` modifiers of the called HELLUM function.

The output of the compiler on the `Test` contract is displayed in Figure 6, where we use the concrete ILLUM syntax supported by the compiler. There, we can observe how the clause `f_run` contains a process with two branches, one for each conditional branch in Figure 5. Both of these branches call the `Check` clause so to enable the whole `call` if and only if the corresponding conditional branch in HELLUM would be taken. The clause `Check` requires in its precondition that its argument is true, so blocking the `f_run` branches which do not correspond to the HELLUM execution. The ILLUM branches call clause `Pay` to transfer the assets according to the `a.transfer(...)` commands found in the corresponding conditional branch of `f` in Figure 5. Finally, each branch calls `f_next` passing the updated state in the parameters.

The correctness of the compilation from HELLUM to ILLUM is straightforward. First, the code transformations used to bring the HELLUM contract in normal form, detailed in Appendix I, are standard and clearly preserve the semantics of contracts. Second, the ILLUM contract clauses are generated precisely following the simple structure of the obtained normal form, so their semantics is faithful to the original code by construction. Indeed, we perform the same conditional checks in ILLUM, we transfer the same tokens, and we we update the state variables in the same way it is done by the simultaneous assignment of the HELLUM normal form.

Evaluation. To evaluate the practicality of ILLUM as a compilation target of higher-level contract languages,

```
clause f_run(x,bal_T; y,a) {
  precond_wallet: bal_T:T
  precond_if: ((bal_T-y>10) && y>20) ||
              (bal_T-y<=10)
  process:
    call( Check(bal_T-y>10) | Pay(a,bal_T-y,T) |
          f_next((bal_T-y)+1,y) )
    call( Check(bal_T-y<=10) | f_next(bal_T-y,bal_T
          ) )
}
clause f_next(x,bal_T; ) {
  precond_wallet: bal_T:T
  precond_if: true
  process:
    call f(x,bal_T)
    call g(x,bal_T)
}
clause g_run(x,bal_T; ) {
  precond_wallet: bal_T:T
  precond_if: true
  process:
    call g_next(x,bal_T)
}
clause g_next(x,bal_T; ) {
  precond_wallet: bal_T:T
  precond_if: true
  process:
    call f(x,bal_T)
    call g(x,bal_T)
}
clause Pay(a,v,t; ) {
  precond_wallet: v:t
  precond_if: true
  process:
    send(v:t -> a)
}
clause Check(b; ) {
  precond_wallet:
  precond_if: b
  process:
    send()
}
```

Figure 6: Translation of the `Test` contract in ILLUM.

we construct a benchmark of smart contracts. The benchmark comprises common use cases, like e.g. those in the OpenZeppelin library of Solidity contracts. Besides that, we also include more complex contracts like those found in DeFi: in particular, we implement a constant-product Automated Market Maker (AMM) and a Lending Pool. All the contracts in our benchmark are implemented in HELLUM, and automatically translated into ILLUM by our prototype compiler. Table 1 shows the size (LoC and bytes) of the HELLUM contracts and of the corresponding ILLUM clauses. Despite the compilation into ILLUM produces just a 2x-3x expansion in the size, the ILLUM code is inherently less readable than the original HELLUM contract, as usual for intermediate-level languages.

8. Related work

Intermediate languages have already been studied that, like our ILLUM, can serve as a compilation target of high-level smart contract languages. SCILLA [26] is an intermediate language that targets *account-based* blockchains and is executed natively by the Zilliqa blockchain. SCILLA has an imperative core featuring loop-free statements (with operators to update state variables and transfer assets), and a higher-order functional core with structural recursion on lists and naturals. This gives a form of iteration, and consequently requires the underlying blockchain to implement a gas mechanism to thwart denial-of-service attacks. Instead, in ILLUM every operation has a *bounded* compu-

tational cost, thus eliminating the need for a gas mechanism. Nonetheless, ILLUM achieves Turing-completeness by spreading complex computations across multiple basic actions. The goals of SCILLA and ILLUM are different: SCILLA is meant to be directly interpreted by blockchain nodes, while ILLUM is meant to be compiled to a lower-level script language, demanding for a weaker runtime support from the underlying blockchain.

In the UTXO realm, a variety of contract languages have been proposed, starting from Bitcoin Script [27], a low-level, stack-based language that is interpreted by Bitcoin nodes. Since writing spending conditions directly in Bitcoin Script can be quite complex, a few languages have been proposed to relieve programmers from this task, like Simplicity [28] and Miniscript [29]. Although these languages allow for representing Bitcoin scripts in a more structured and human-readable manner, they do not make writing contracts in Bitcoin much easier (except for basic single-transaction use cases). In general, Bitcoin contracts take the form of protocols where participants exchange messages and send transactions to the blockchain [30]. The languages [28], [29] however can only specify the individual transactions used in these protocols, and not the overall global contract. BitML [31] is a higher-level language that allows to specify global contracts and compile them to *sets* of Bitcoin transactions. To be compliant with the strict constraints of Bitcoin, the expressive power of BitML is limited to contracts with *bounded* execution lengths. This rules out relevant use cases, like e.g. the auction in Section 2 and the crowdfunding in Section 7, which allow for an unbounded number of steps. The work [32] enhances the expressiveness of BitML with a weak form of recursion: each recursive step can only be performed with the approval of *all* participants. In ILLUM instead recursion is unconstrained: participants cannot prevent an enabled recursion step from happening. This expressiveness gain comes at a cost, in that ILLUM cannot be compiled into standard Bitcoin transactions. Executing ILLUM on Bitcoin would be possible by extending Bitcoin Script with covenants, in a form that is just a bit more expressive than a recently proposed covenant opcode [33].

To overcome the expressiveness limitations of the Bitcoin UTXO model, the Cardano blockchain extends it with some additional functionalities [34], [35]: (i) special transaction fields to store contract state; (ii) a mechanism to preserve contract code along chains of transactions; (iii) native custom tokens [36]; (iv) an expressive scripting language [14]. The first three functionalities are present also in our UTXO model: in particular, we use arg fields to encode the contract state, and covenants to preserve the contract code. The main difference between our UTXO model and Cardano's is the scripting language. Cardano's scripting language is an untyped lambda calculus enriched with built-in functions to interact with the blockchain. This makes Cardano scripts Turing-complete, and consequently requires a complex runtime environment (including a gas mechanism). Our scripts instead are not Turing-complete, but still our contracts are such, as shown in Section 3. Existing smart contract languages for Cardano (e.g., Plutus, Aiken), although based on high-level languages (i.e., Haskell), impose a *low-level* programming style for smart contracts, requiring developers to reason at the level of transactions, not too distantly from the awkward UTXO programming style exemplified in Section 2.2. Programming in this style is inherently more complex than using higher-level procedural languages, which are mainstream in the blockchain developers community. Indeed, in existing Cardano languages, performing a contract action amounts to replacing the old state with a new one (i.e., spending some transaction outputs with a new transaction). Accordingly, programming a contract action amounts to verifying through the redeem scripts that the new state is a correct update of the old one, checking multiple transaction fields that encode the contract state. This programming style is quite burdensome, since forgetting even a single check may give rise to vulnerabilities (e.g., adversaries could be able to set a data field of the new state to a value at their choice). To the best of our knowledge, we are the first to propose a practical procedural high-level language for smart contracts that can be automatically compiled to UTXO blockchains.

Our UTXO model can be implemented efficiently. Most operators of our scripting language are borrowed from Bitcoin Script, which is interpreted very efficiently by Bitcoin nodes. Implementing arg fields and the opcodes to access them poses no challenge. Covenants, both of kind verscr and verrec, can be implemented by exploiting a mechanism similar to "Pay to Script Hash" in Bitcoin [37], which stores in the scr field the hash of the script, instead of the script itself. For the verscr covenant, we would specify the hash h of the script (rather than the script) in the first argument: then, $\mathsf{verscr}(h, i)$ would simply check that the hash in $\mathsf{rtxo}(i).\mathsf{scr}$ is equal to h. Similarly, the $\mathsf{verrec}(i)$ covenant would check that the hash in $\mathsf{rtxo}(i).\mathsf{scr}$ is equal to the hash in the current script, $\mathsf{ctxo}.\mathsf{scr}$. Both checks can be done very efficiently, as one just needs to compare two hashes. A further optimization can be achieved by exploiting Taproot [38], a mechanism allowing users to reveal the parts of the contract (clause branches) only when they are executed. This decreases the size of witnesses that must be included along with transactions, which in turn decreases the transaction fees.

One of the main advantages of UTXO blockchains over account-based ones is the possibility of parallelizing transaction validation over multiple cores. Indeed, there is an easy criterion to determine if two UTXO transactions are parallelizable, i.e. checking that their inputs are disjoint. Instead, in account-based blockchains two transactions, even targeting different contracts, may read-/write the same part of the state, e.g. when they update the same account. A few works study how to overcome this limitation: some of them exploit dynamic techniques adopted from software transactional memory [6], [39]–[41], while some others are based on the static analysis of contracts [7], [42]. In particular, [6] provides empirical evidence about the effectiveness of parallelizing transaction execution in Ethereum, showing an overall speedup of 1.33x for miners and 1.69x for validators, using only three cores, based on a benchmark of representative contracts.

Acknowledgments. Work partially supported by projects PRIN 2022 DeLiCE (F53D23009130001) and SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

References

[1] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "SoK: Decentralized Finance (DeFi)," 2021.

[2] S. Wang, W. Ding, J. Li, Y. Yuan, L. Ouyang, and F. Wang, "Decentralized Autonomous Organizations: Concept, model, and applications," *IEEE Trans. Comput. Soc. Syst.*, vol. 6, no. 5, pp. 870–878, 2019.

[3] "Defillama: Total value locked," https://defillama.com, 2024.

[4] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Financial Cryptography Workshops*, ser. LNCS, vol. 10323. Springer, 2017, pp. 478–493.

[5] L. Brünjes and M. J. Gabbay, "UTxO- vs account-based smart contract blockchain programming paradigms," in *ISoLA*, ser. LNCS, vol. 12478. Springer, 2020, pp. 73–88.

[6] T. D. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2017, pp. 303–312.

[7] M. Bartoletti, L. Galletta, and M. Murgia, "A theory of transaction parallelism in blockchains," *Log. Methods Comput. Sci.*, vol. 17, no. 4, 2021.

[8] S. Eskandari, S. Moosavi, and J. Clark, "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain," in *Financial Cryptography*. Springer, 2020, pp. 170–189.

[9] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *IEEE Symp. on Security and Privacy*. IEEE, 2020, pp. 910–927.

[10] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *IEEE Symp. on Security and Privacy*. IEEE, 2022, pp. 198–214.

[11] "Flashbots transparency dashboard: REV activities since the Merge," 2023, available at https://transparency.flashbots.net/. Accessed: 2023-09-20.

[12] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of Bitcoin transactions," in *Financial Cryptography*, ser. LNCS, vol. 10957. Springer, 2018, pp. 541–560.

[13] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, "Developing secure Bitcoin contracts with BitML," in *ESEC/FSE*, 2019.

[14] Plutus Team, "Formal specification of the PlutusCore language," 2022. [Online]. Available: https://aiken-lang.org/resources/plutus-core-specification.pdf

[15] D. Perez and B. Livshits, "Broken Metre: Attacking resource metering in EVM," in *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2020.

[16] M. Bartoletti, S. Lande, and R. Zunino, "Bitcoin covenants unchained," in *ISoLA*, ser. LNCS, vol. 12478. Springer, 2020, pp. 25–42.

[17] M. Möser, I. Eyal, and E. G. Sirer, "Bitcoin covenants," in *Financial Cryptography Workshops*, ser. LNCS, vol. 9604. Springer, 2016, pp. 126–141.

[18] R. O'Connor and M. Piekarska, "Enhancing Bitcoin transactions with covenants," in *Financial Cryptography Workshops*, ser. LNCS, vol. 10323. Springer, 2017.

[19] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 125:1–125:36, 2019.

[20] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *IEEE Computer Security Foundations Symposium (CSF)*, 2019, pp. 256–271.

[21] M. Bartoletti, R. Marchesin, and R. Zunino, "Secure compilation of rich smart contracts on poor UTXO blockchains," *CoRR*, vol. abs/2305.09545, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.09545

[22] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg, "Counter machines and counter languages," *Mathematical systems theory*, vol. 2, no. 3, pp. 265–283, 1968.

[23] M. Bartoletti, S. Lande, and R. Zunino, "Computationally sound Bitcoin tokens," in *IEEE Computer Security Foundations Symposium (CSF)*, 2021, pp. 1–15.

[24] Solidity Academy, "#100DaysOfSolidity #073: Understanding denial of service attacks in Solidity smart contracts," 2023. [Online]. Available: https://medium.com/@solidity101/100daysofsolidity-073-understanding-denial-of-service-attacks-in-solidity-smart-contracts-a790de3d0943

[25] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust*, ser. LNCS, vol. 10204. Springer, 2017, pp. 164–186.

[26] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with Scilla," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 185:1–185:30, 2019.

[27] Bitcoin Wiki, "Bitcoin Script," https://en.bitcoin.it/wiki/Script, 2014.

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

[29] P. Wuille and A. Poelstra, "Miniscript: Streamlined Bitcoin scripting," https://medium.com/blockstream/miniscript-bitcoin-scripting-3aeff3853620, 2019.

[30] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino, "SoK: unraveling Bitcoin smart contracts," in *POST*, ser. LNCS, vol. 10804. Springer, 2018, pp. 217–242.

[31] M. Bartoletti and R. Zunino, "BitML: a calculus for Bitcoin smart contracts," in *ACM CCS*, 2018.

[32] M. Bartoletti, S. Lande, M. Murgia, and R. Zunino, "Verifying liquidity of recursive Bitcoin contracts," *Log. Methods Comput. Sci.*, vol. 18, no. 1, 2022.

[33] J. Rubin, "CHECKTEMPLATEVERIFY," 2020, BIP 119, https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki.

[34] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler, "The extended UTXO model," in *Financial Cryptography and Data Security Workshops*, ser. LNCS, vol. 12063. Springer, 2020, pp. 525–539.

[35] Cardano, "EUTXO handbook," https://ucarecdn.com/3da33f2f-73ac-4c9b-844b-f215dcce0628/EUTXOhandbook_for_EC.pdf, 2022.

[36] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. P. Jones, P. Vinogradova, P. Wadler, and J. Zahnentferner, "UTXO$_{\mathrm{ma}}$: UTXO with multi-asset support," in *ISoLA*, ser. LNCS, vol. 12478. Springer, 2020, pp. 112–130.

[37] G. Andresen, "Pay to Script Hash," 2012, BIP 16, https://github.com/bitcoin/bips/wiki/Comments:BIP-0016.

[38] A. T. Pieter Wuille, Jonas Nick, "Taproot: SegWit version 1 spending rules," 2020, BIP 341, https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

[39] T. D. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," *Bulletin of the EATCS*, vol. 124, 2018.

[40] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing (PDP)*, 2019, pp. 83–92.

[41] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in Ethereum smart contracts," in *Tokenomics*, ser. OASIcs, vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 4:1–4:15.

[42] G. Pîrlea, A. Kumar, and I. Sergey, "Practical smart contract sharding with ownership and commutativity analysis," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2021, pp. 1327–1341.

[43] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1988, pp. 12–27.

A. Symbolic model of ILLUM contracts

In this appendix we fully define the symbolic model. We start with the syntax of contracts, clauses and configurations. We will then define the semantics of ILLUM as a state transition system in Figure 8.

Notation. To improve readability, in the appendices we slightly simplify the model presented in the main text. First, we assume a single type of token. From a technical standpoint, handling multiple tokens would just require to change the semantics so that sums of values become sums of sequences of tokens. We prefer to omit this, as it would bloat an already heavy notation. We also simplify the arithmetic of the blockchain. In particular, we assume integers to be the only numerical data type. This restricts the arithmetic operations that are possible in contracts. Again, having rationals and divisions can be done without changing the fundamental theory developed in these appendices. We also omit mappings, since they are not strictly needed in the definition of the compiler, and could be easily added to the model. Lastly, we adopt a different notation in the naming of the internal and external parameters of a clause: instead of the In and Ex, hereafter we use α and β. Actual values passed to clauses are also changed from $\overline{\text{In}}$ to a and from $\overline{\text{Ex}}$ to b.

Syntax of expressions. Before introducing the terms of ILLUM's symbolic model, we define the syntax of expressions. First we have arithmetic expressions $\mathcal{E}$, defined as:

$$\begin{array}{lll} \mathcal{E} ::= & k & \text{(constants)} \\ & \mid \alpha, \beta & \text{(variables)} \\ & \mid |\mathcal{E}| & \text{(size)} \\ & \mid \mathcal{E} + \mathcal{E} & \\ & \mid \mathcal{E} - \mathcal{E} & \\ & \mid H(\mathcal{E}) & \text{(hash)} \\ & \mid \mathsf{if}\ p\ \mathsf{then}\ \mathcal{E}\ \mathsf{else}\ \mathcal{E} & \end{array}$$

Then there are name expressions $\mathcal{N}$, defined as:

$$\mathcal{N} ::= \mathsf{A}\ \text{(names)} \mid \alpha, \beta\ \text{(variables)}$$

We also define boolean expressions, or conditions as

$$p ::= true \mid \mathsf{not}\ p \mid p\ \mathsf{and}\ p \mid \mathcal{E} = \mathcal{E} \mid \mathcal{N} = \mathcal{N} \mid \mathcal{E} < \mathcal{E}$$

In the following, we will also freely use other boolean operations that can be derived from the ones listed above.

Definition 4 (Contracts)**.** The syntax of contracts is:

$C ::= \sum_{i \in I} D_i$	contract
$D ::=$	contract branch
$\mathtt{call}\ (\cdots, \mathtt{X}_i\langle \boldsymbol{\mathcal{P}_i}; ? \rangle, \cdots)$	call to clauses $\mathtt{X}_1 \cdots \mathtt{X}_n$
$\mid \mathtt{send}\ (\cdots, \mathcal{E}_i \to \mathcal{N}_i \cdots)$	transfer $\mathcal{E}_i$ to each $\mathcal{N}_i$
$\mid \mathcal{N} : D$	wait for $\mathcal{N}$ authorization
$\mid \mathtt{after}\ \mathcal{E} : D$	wait until time $\mathcal{E}$
$\mid \mathtt{afterRel}\ \mathcal{E} : D$	wait $\mathcal{E}$ after activation

where $\boldsymbol{\mathcal{P}_i}$ is a sequence of arithmetic expressions $\mathcal{E}$ and name expressions $\mathcal{N}$. We also assume that:

(i) each recursion variable has a unique defining equation $\mathtt{X}(\boldsymbol{\alpha}; \boldsymbol{\beta}) = \{\mathcal{E}\ \mathtt{if}\ p\}\ C$, with the syntax below;
(ii) the sequence of expressions $\boldsymbol{\mathcal{P}_i}$ passed to a called clause $\mathtt{X}$ matches, in length and typing, the sequence of formal parameter $\boldsymbol{\alpha}$ of formal parameters;
(iii) the order of decorations is immaterial, for instance $\mathsf{A} : \mathtt{after}\ t : D$ is identified with $\mathtt{after}\ t : \mathsf{A} : D$.

Definition 5 (Clauses)**.** A clause is defined by an equation

$$\mathtt{X}(\boldsymbol{\alpha}; \boldsymbol{\beta}) = \{\mathcal{E}\ \mathtt{if}\ p\}\ C.$$

where $\{\mathcal{E}\ \mathtt{if}\ p\}$ is the *funding precondition* and C is a contract. The clause takes two sequences of parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Parameters are of two types: integers and participants, and we will assume that in the sequences all integer parameters precede participants. We ask that the only variables in $\mathcal{E}$, and in all the expressions contained in C and p, are the ones taken as parameters by $\mathtt{X}$.

The term $\{\mathcal{E}\ \mathtt{if}\ p\}$ gives conditions that must hold in order to activate C. In particular, $\mathcal{E}$ denotes the amount of tokens that must be stored in C. These tokens will be taken both from the calling contract, and from additional deposits. The proposition p is a predicate on the actual values that are passed to the clause at call time. If p is not satisfied then the clause cannot be called, and C will not be activated. When $p = true$, we simply write $\{\mathcal{E}\}\ C$.

Evaluation and closed form. We specify below the substitution of actual values for parameters. By substituting the parameters of a clause $\mathtt{X}$ with two sequences of actual values $\boldsymbol{a}$ and $\boldsymbol{b}$ (with $a_i \in \mathbb{Z} \cup \mathsf{Part}$, and $b_i \in \mathbb{Z} \cup \mathsf{Part} \cup \{*\}$) we produce an *instantiated clause*, denoted with $\mathtt{X}\langle \boldsymbol{a}; \boldsymbol{b} \rangle$. We define a relation $\mathtt{X}\langle \boldsymbol{a}; \boldsymbol{b} \rangle \equiv \{v\}\ C'$ that holds iff no b_j is equal to $*$ and the following conditions hold:

(i) The actual values are well-typed, i.e. the types of $\boldsymbol{a}$ and $\boldsymbol{b}$ match the ones of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ respectively. In particular, there must be $*$ among the elements of $\boldsymbol{b}$.
(ii) $[\![\mathcal{E}\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = v$;
(iii) $[\![p\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = true$;
(iv) $[\![C\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = C'$.

Here, writing $\boldsymbol{a}/\boldsymbol{\alpha}$ means that we replace every parameter α_i in the expression with the value a_i, and $[\![\cdot]\!]$ is a simple evaluation operator that performs all arithmetic and logic operations present in an expression. Notice that, after the evaluation, every expression inside C' is reduced to an constant. Such a contract is said to be in *closed form*. Unless specified otherwise, from this point onward we will be working with contracts in closed form.

Definition 6 (Configurations)**.** A configuration Γ is a term $\tilde{\Gamma} \mid t \mid \mathcal{D}(w)$, where $t \in \mathbb{N}$ denotes the time, $\mathcal{D}(w)$ is the destroyed funds counter, and $\tilde{\Gamma}$ has the following syntax:

$\tilde{\Gamma} ::= \emptyset$	empty
$\mid \langle C, v \rangle_x^t$	active contract
$\mid \langle \mathsf{A}, v \rangle_x$	deposit
$\mid \Phi$	complete advertisement
$\mid \Theta$	incomplete advertisement
$\mid \mathsf{A}[\chi]$	authorization
$\mid \tilde{\Gamma} \mid \tilde{\Gamma}$	parallel composition

We also assume that (i) parallel composition is associative and commutative; (ii) all parallel terms are distinct and names are never repeated; (iii) all contracts C appearing in a configuration are in closed form.

$$\frac{\Gamma \text{ contains } \langle \mathsf{A}, v_1 \rangle_{z_1} \text{ and } \langle \mathsf{A}, v_2 \rangle_{z_2}}{\Gamma \xrightarrow{auth-join(\mathsf{A},z_1,z_2,i)} \Gamma \mid \mathsf{A}[z_1, z_2, i \rhd v_1 + v_2]}$$

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

$$\frac{\Gamma \text{ contains } \langle \mathsf{A}, v + v' \rangle_z}{\Gamma \xrightarrow{auth-divide(\mathsf{A},z,v,v')} \Gamma \mid \mathsf{A}[z \rhd v, v']}$$

$$\frac{\Gamma = \Gamma' \mid \mathsf{A}[z \rhd v, v'] \quad y, y' \text{ fresh}}{\Gamma \mid \langle \mathsf{A}, v + v' \rangle_z \xrightarrow{divide(z,v,v')} \Gamma' \mid \langle \mathsf{A}, v \rangle_y \mid \langle \mathsf{A}, v' \rangle_{y'}}$$

$$\frac{\Gamma \text{ contains } \langle \mathsf{A}, v \rangle_z}{\Gamma \xrightarrow{auth-donate(\mathsf{A},z,\mathsf{B})} \Gamma \mid \mathsf{A}[z \rhd \mathsf{B}]}$$

$$\frac{\Gamma = \Gamma' \mid \mathsf{A}[z \rhd \mathsf{B}] \quad y \text{ fresh}}{\Gamma \mid \langle \mathsf{A}, v \rangle_z \xrightarrow{donate(z,\mathsf{B})} \Gamma' \mid \langle \mathsf{B}, v \rangle_y}$$

Figure 7: Semantics of ILLUM deposits.

Active contracts. An *active contract* is a term $\langle C, v \rangle_x^t$. It is uniquely identified by its name x, and it represents an amount of v tokens (its *balance*) that can only be spent according to the conditions set by one of C's branches. The integer t is the time when the contract has been added to the configuration. We assume C to be in closed form.

Deposits. The terms $\langle \mathsf{A}, v \rangle_x$ in a configuration, called *deposits*, are uniquely identified by their name x, and represent an amount v of tokens owned by participant A. The owner of a deposit is the only one who can provide the authorization to spend it. Figure 7 defines the semantics of deposits: $\langle \mathsf{A}, v \rangle_x$ can be donated to another participant, split into two smaller deposits, or merged with another one. Moreover, a deposit can be spent to fund the activation of a new contract, or the execution of a contract step. Deposit can be destroyed. In this case the destroyed tokens are added to a counter $\mathcal{D}(w)$, which keeps track of the tokens that was stored in a deposit which the owner has decided to destroy, denoting with $w \in \mathbb{N}$ the total. We assume that only dishonest participants can spend the tokens in the counter, and that they can do so freely, without the need to produce any authorization term.

Advertisements (general). Some actions, in particular the ones related to contracts, require to be advertised before they can be performed, meaning that a participants who wants to execute them has to inform the others by introducing an *advertisement term* in the configuration. Such terms can be of two kinds: either *complete* or *incomplete*. An incomplete advertisement Θ is only used as a message, while the complete term Φ is also needed by the semantics in order to carry on certain actions, as we will see in the next section. As a result, incomplete advertisement have the option of leaving some features unspecified. The actions that can be advertised are the following: (i) the activation of a new contract; (ii) the continuation of an active contract; (iii) the destruction of a set of deposits. In the following paragraphs we will describe precisely how each of the three term is structured. The main focus will be the definition of complete advertisement, and, after that, we will mention what are the parts can be left unspecified to obtain the incomplete version.

Advertisement (initial). A complete initial advertisement is a term $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h$, where $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle$ presents the proposed contract and its parameters, $\boldsymbol{z}$ is a non empty list of deposit names (that will be spent to fund the initialization), and $w \in \mathbb{N} \cup \{\star\}$ is the amount of destroyed funds that are going to be taken from $\mathcal{D}$ and used in the initialization. Here $\star$ is a special symbol that means no currency is taken from the counter: it will be treated as 0 in arithmetical operations [2]. The subscript $h \in \mathbb{N}$ is just a nonce, used to differentiate two otherwise identical terms. In an initial advertisement the clause $\mathtt{X}$ must have its precondition p equal to $true$. This is a technical requirement, added to simplify the language implementation, but it can also be justified intuitively: since the contract is not yet started, if the participants want the arguments to satisfy some proposition p, they can simply choose to initiate a different contract. In a complete advertisement the special symbol $*$ must not appear in the parameters β passed to the clause.

Advertisements (continuation). When a configuration contains an active contract $\langle C, v \rangle_x^t$, a participant that wants to execute D, the j-th branch of x, will produce the advertisement term $\Phi = [\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)]_h$. Like in the previous case, $\boldsymbol{z}$ and w are used to specify the source of additional funds used in the continuation action (here we also allow $\boldsymbol{z}$ to be an empty list); and $h \in \mathbb{N}$ is again a nonce. The term $\bar{D}$ is called *advertised branch*, and it needs a more detailed presentation. $\bar{D}$ is constructed by taking a branch D while replacing the question marks $?$ inside a `call` termination with actual values b_j which can be freely chosen by the participant producing the advertisement term. We will identify $\bar{D}$ and $\bar{D}'$ if one can be obtained from the other by exchanging the decorations' order. Notice that, since every active contract is in closed form, the only expressions appearing inside $\bar{D}$ will be constants. To denote that $\bar{D}$ has been constructed starting from D we write $\bar{D} \approx D$. The terms $\bar{D}$ have the syntax:

$$\begin{aligned}\bar{D} ::= {} & \mathtt{send}\ (v_1 \to \mathsf{A}_1 \cdots v_n \to \mathsf{A}_n) \\ \mid {} & \mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle \cdots \mathtt{X}_n\langle \boldsymbol{a_n}\,;\boldsymbol{b_n}\rangle) \\ \mid {} & \mathsf{A} : \bar{D} \mid \mathtt{after}\, t : \bar{D} \mid \mathtt{afterRel}\, \delta : \bar{D}\end{aligned}$$

with $a_j^i \in \mathbb{Z} \cup \mathsf{Part}$ and $b_j^i \in \mathbb{Z} \cup \mathsf{Part} \cup \{*\}$.

Advertisements (destruction). A destruction advertisement $\Phi = [\boldsymbol{z}\,;w]_h$ is produced when one wants to remove some deposits from the configuration, adding their value to the destroyed funds counter. Here $\boldsymbol{z}$ is a nonempty list of the deposit names that are going to be destroyed, $w \in \mathbb{N} \cup \{\star\}$ is an amount of funds from the counter, h is a nonce that differentiates two otherwise identical terms.

2. In a configuration an advertisement term with $w = \star$ behaves identically to one with $w = 0$. The only difference between the two terms is that honest participants will only be allowed to produce terms that have $w = \star$. We address the reason behind the use of $\star$ when comparing the symbolic model with the computational one.

Incomplete advertisements. In an incomplete advertisement term Θ some informations may be left unspecified. Again, we have three types of advertisements: (i) Initial, with $\Theta = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]$. Here some of the values β_i may be equal to $*$, the sequence $\boldsymbol{z}$ may be empty, and w can also take the value $*$. (ii) Continuation, with $\Theta = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]$. Similarly to the case above, w can be set equal to $*$, and values b_j inside `call` operations in $\bar{D}$ can also be set to $*$. (iii) Destruction, with $\Theta = [\boldsymbol{z}\,;w]$, where we allow w to be equal to $*$.

Validity of advertisements. We now define *validity* of an advertisement, which must hold for an advertised operation to be performed. Mainly, validity checks that all the terms in the advertisement actually occur in the configuration. Remember that the value $\star$ is treated as a 0 in all arithmetic operations. A complete advertisement Φ is *valid in* Γ if one of the following holds:

- (Initial) $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h$, where $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle \equiv \{v\}\ C$, and the configuration Γ contains deposits $\langle \mathsf{A}_j, u_j\rangle_{z_j}$ and the counter $\mathcal{D}(w')$, with $w' \geq w$ and $\sum_j u_j + w \geq v \geq 0$.
- (Continuation) $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h$, and the following conditions hold: (i) the configuration contains the deposits $\langle \mathsf{A}_j, u_j\rangle_{z_j}$, the contract $\langle C, v\rangle_x^t$, and $\mathcal{D}(w')$ with $w' \geq w$; (ii) The j-th branch of C is a D such that $\bar{D} \approx D$; (iii) the time t in Γ is greater than all t_i appearing in `after` decorations of $\bar{D}$, and $t - t_0$ is greater than all δ_i appearing in `afterRel` decorations of $\bar{D}$; (iv) if $\bar{D}$ ends in `send` $(v_1 \to \mathsf{A}_1, \cdots v_n \to \mathsf{A}_n)$ then we must have $\sum_j u_j + w + v \geq \sum_i v_i$; (v) if instead $\bar{D}$ ends in `call` $(\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a_n}\,;\boldsymbol{b_n}\rangle)$ then there must be $v_i \geq 0$, and C_i such that for every $i = 1 \cdots n$ we have $\mathtt{X}_i\langle \boldsymbol{a_i}\,;\boldsymbol{b_i}\rangle \equiv \{v_i\}\ C_i$, and $\sum_j u_j + w + v \geq \sum_i v_i$;
- (Destruction) $\Phi = [\boldsymbol{z}\,;w]_h$ and the configuration Γ contains the deposits z_j, and $\mathcal{D}(w')$ with $w' \geq w$.

Authorizations. Authorization are terms of the form $\mathsf{A}[\chi]$, where A is the authorizer, and $\chi = \cdots \rhd \cdots$ has a LHS that denotes what is being authorized, and a RHS denoting the authorized action. Authorizations are required for all deposit actions (joining, dividing or donating deposit), and for spending the deposits in an advertised action. Lastly, some contract branches may require a participant authorization:

1) $z \rhd \Phi$, where z is a deposit, is used to authorize the use of z to fund the action advertised by Φ.
2) $x \rhd \Phi$, where x is a contract and $\Phi = \left[\mathsf{A} : \bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h$, is used to authorize the execution of the j-th branch of x, satisfying the decoration $\mathsf{A} : \cdots$.
3) $z_1, z_2, i \rhd v_1 + v_2$ is used to authorize the use of deposit $\langle \mathsf{A}, v_i\rangle_{z_i}$ in a join operation with another deposit z_j of value v_j (we have $i \neq j$ and $i, j \in \{1, 2\}$).
4) $z \rhd v, v'$ is used to authorize the splitting of deposit $\langle \mathsf{A}, v + v'\rangle_z$ into two, of value v and v' respectively.
5) $z \rhd \mathsf{B}$ is used to authorize the transfer of deposit $\langle \mathsf{A}, v\rangle_z$ to a participant B. If $\chi = z \rhd \Phi$, the authorization allows to use the deposit z to fund the action advertised by Φ.

Time. A configuration Γ keeps track of time by simply including an integer t. This term is used to check whether a branch of an active contract decorated by `after` or `afterRel` can be executed.

Definition 7 (Semantics)**.** The operational semantics of ILLUM is a labelled transition system between configurations, defined by the rules in Figure 8.

$$\frac{\Gamma \text{ does not contain } \Theta}{\Gamma \xrightarrow{msg(\Theta)} \Gamma \mid \Theta}$$

$$\frac{\Phi \text{ valid in } \Gamma \quad \Gamma \text{ does not contain } \Phi}{\Gamma \xrightarrow{adv(\Phi)} \Gamma \mid \Phi}$$

$$\frac{\Gamma \text{ contains } \Phi \text{ but not } \mathsf{B}[z \rhd \Phi] \quad \Phi \text{ valid in } \Gamma \text{ and funded with deposit } z}{\Gamma \xrightarrow{auth-in(\mathsf{B},z,\Phi)} \Gamma \mid \mathsf{B}[z \rhd \Phi]}$$

$$\frac{\Gamma \text{ contains } \Phi \text{ but not } \mathsf{A}[x \rhd \Phi] \quad \Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h \text{ valid in } \Gamma \quad \bar{D} \approx \mathsf{A} : D'}{\Gamma \xrightarrow{auth-act(\mathsf{A},\Phi)} \Gamma \mid \mathsf{A}[x \rhd \Phi]}$$

$$\frac{\begin{array}{c}\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h \text{ valid in } \Gamma' \quad \mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle \equiv \{v\}\ C \\ \Delta_{pre} = \Phi \mid \left(\|_j \langle \mathsf{B}_j, u_j\rangle_{z_j}\right) \mid \left(\|_j \mathsf{B}_j[z_j \rhd \Phi]\right) \\ \Gamma' = (\Gamma \mid \Delta_{pre} \mid \mathcal{D}(w') \mid t)\end{array}}{\xrightarrow{init(\Phi,x)} \quad \Gamma \mid \langle C, v\rangle_x^t \mid \mathcal{D}((w' - w)) \mid t}$$

$$\frac{\begin{array}{c}D = \mathsf{A}_1 : \cdots : \mathsf{A}_n : D' \qquad D' \neq \mathsf{A} : D'' \\ \Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h \text{ valid in } \Gamma' \text{ with } \bar{D} \approx D \\ \bar{D} \text{ ends in } \mathtt{send}\ (v_1 \to \mathsf{C}_1 \cdots v_m \to \mathsf{C}_m) \\ \Delta_{auth} = \left(\|_l \mathsf{B}_l[z_l \rhd \Phi]\right) \mid \left(\|_k \mathsf{A}_k[x \rhd \Phi]\right) \\ \text{(if } \mathsf{A}_k = \mathsf{A}_{k'} \text{ the authorization only appears once)} \\ \Delta_{dep} = \left(\|_l \langle \mathsf{B}_l, u_l\rangle_{z_l}\right) \quad \Delta_{post} = \left(\|_i \langle \mathsf{C}_i, v_i\rangle_{y_i}\right) \\ \Delta_{pre} = \Phi \mid \Delta_{auth} \mid \Delta_{dep} \mid \langle C, v\rangle_x^t \quad y_1 \cdots y_m \text{ fresh} \\ \Gamma' = (\Gamma \mid \Delta_{pre} \mid \mathcal{D}(w'))\end{array}}{\xrightarrow{send(\Phi)} \quad \Gamma \mid \Delta_{post} \mid \mathcal{D}((w' - w))}$$

$$\frac{\begin{array}{c}D = \mathsf{A}_1 : \cdots : \mathsf{A}_n : D' \qquad D' \neq \mathsf{A} : D'' \\ \Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h \text{ valid in } \Gamma' \text{ with } \bar{D} \approx D \\ \bar{D} \text{ ends in } \mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle \cdots \mathtt{X}_m\langle \boldsymbol{a_m}\,;\boldsymbol{b_m}\rangle) \\ \text{and } \forall i.\ \mathtt{X}_i\langle \boldsymbol{a_i}\,;\boldsymbol{b_i}\rangle \equiv \{v_i\}\ C_i \\ \Delta_{auth} = \left(\|_l \mathsf{B}_l[z_l \rhd \Phi]\right) \mid \left(\|_k \mathsf{A}_k[x \rhd \Phi]\right) \\ \text{(if } \mathsf{A}_k = \mathsf{A}_{k'} \text{ the authorization only appears once)} \\ \Delta_{dep} = \left(\|_l \langle \mathsf{B}_l, u_l\rangle_{z_l}\right) \quad \Delta_{post} = \left(\|_i \langle C_i, v_i\rangle_{y_i}^t\right) \\ \Delta_{pre} = \Phi \mid \Delta_{auth} \mid \Delta_{dep} \mid \langle C, v\rangle_x^{t_0} \quad y_1 \cdots y_m \text{ fresh} \\ \Gamma' = (\Gamma \mid \Delta_{pre} \mid \mathcal{D}(w') \mid t)\end{array}}{\xrightarrow{call(\Phi)} \quad \Gamma \mid \Delta_{post} \mid \mathcal{D}((w' - w)) \mid t}$$

$$\frac{\delta > 0}{\Gamma \mid t \xrightarrow{delay(\delta)} \Gamma \mid t + \delta}$$

Figure 8: Semantics of ILLUM contracts.

$e ::= v$	integer constant
$\mid e \circ e'$	binary operations ($\circ \in \{+, -, =, <\}$)
$\mid e.n$	n-th element of a list
$\mid$ rtxw	witnesses of the redeeming tx input
$\mid \lvert e \rvert$	size (in bytes)
$\mid H(e)$	hash [3]
$\mid$ if e then e' else e''	conditional check
$\mid$ versig(e, e')	signature verification
$\mid$ absAfter $e : e'$	absolute time constraint
$\mid$ relAfter $e : e'$	relative time constraint
$\mid$ o.f	field f $\in \{$arg, val$\}$
	of o $\in \{$ctxo, rtxo$(e)\}$
$\mid$ inidx	index of redeeming tx input
$\mid$ inlen(tx)	number of inputs of tx $\in \{$rtx, ctx$\}$
$\mid$ outlen(tx)	number of outputs of tx $\in \{$rtx, ctx$\}$
$\mid$ verscr(e, e')	checks if rtxo(e').scr $= e$
$\mid$ verrec(e)	checks if rtxo(e).scr $=$ ctxo.scr

Figure 9: Syntax of scripts.

B. Computational Model

In this appendix we present in more detail low level model of the blockchain that serves as target of compilation for ILLUM contracts.

Definition 8 (Transaction)**.** A transaction T is defined as a 5-uple (in, wit, out, absLock, relLock) where

- in is the list of inputs. Each element of in is a pair (T', i), where T' is a transaction and i is an integer.
- wit is the list of witnesses. It has the same length as in, and each element of wit is a list of integers.
- out is the list of outputs. Each output is a triple (val, scr, arg), where arg is a list of integers, scr is a script (its syntax will be specified in the next paragraphs), and val is an integer.
- absLock is the absolute timelock, and it is a non negative integer.
- relLock is the list of relative timelocks. It has the same lenth as in and each of its elements is a non negative integer.

Given $\mathsf{l} \in \{$in, wit, out, relLock$\}$, we will use $\mathsf{l}(j)$ to denote its j-th element. The lists in, wit, and relLock may be empty; if that is the case we denote them with $\perp$ and we talk about an *initial* (or *coinbase*) transaction.

Definition 9 (Syntax of scripts)**.** The scr field of a transaction output has the syntax in Figure 9. There, the terms ctx and rtx are used to denote the current and redeeming transaction respectively. Moreover, the term ctxo is used by a script to refer to the current output (i.e. the one of which it is the script); and the term rtxo(n), refers to the n-th output of the redeeming transaction. In scr use some shorthands for common logical operations, setting (i) $true \triangleq 1$, (ii) $false \triangleq 0$, (iii) e and $e' \triangleq$ if e then e' else $false$, (iv) not $e \triangleq$ if e then $false$ else $true$, (v) e or $e' \triangleq$ if e then $true$ else e'.

Script evaluation. In order to determine if a transaction can redeem an output, its script must be executed. For this reason, we need to define an evaluation semantics for scripts. We let $[\![\cdot]\!]_{(\mathsf{T},i)}$ be the evaluation operator, where T is the redeeming transaction and i index of the redeeming input. For ease of notation, in the following paragraphs we will shorten $[\![\cdot]\!]_{(\mathsf{T},i)}$ to $[\![\cdot]\!]$, and implicitly assume that $\mathsf{T}.\mathsf{in}(i)$ is the redeeming input, unless specified otherwise. We will also assume that the transaction that is being redeemed is named T', and that the redeemed output is its j-th. Note that T' and j can be determined from T and i, since we have that

$$\mathsf{T}' = \mathit{first}(\mathsf{T}.\mathsf{in}(i)) \quad \text{and} \quad j = \mathit{second}(\mathsf{T}.\mathsf{in}(i)).$$

The evaluation yield $\perp$ when a failure occurs (i.e. trying to access the n-th element of a list with less than n terms). All the operators in the script's syntax are *strict*, meaning that their evaluation yields $\perp$ if one of their arguments is $\perp$. Here we highlight the behaviour of the evaluation semantics on the non-trivial terms:

- $[\![$rtxw$]\!]$ evaluates to $\mathsf{T}.\mathsf{wit}(i)$, which is the sequence of witnesses associated to the redeeming input.
- $[\![$versig$(e, e')]\!]$ evaluates to $true$ if the signature $[\![e']\!]$ is correctly verified on the hash of T^* (which denotes the transaction obtained by replacing the wit field in T with $\perp$) against the key $[\![e]\!]$, and to $false$ otherwise.
- $[\![$absAfter $e : e']\!]$ evaluates to $[\![e']\!]$ if the field absLock of the redeeming transaction T is greater or equal to $[\![e]\!]$, otherwise it evaluates to $\perp$. Similary, $[\![$relAfter $e : e']\!]$ evaluates to $[\![e']\!]$ if the field relLock(i) of T is greater or equal to $[\![e]\!]$, otherwise it fails, yielding $\perp$.
- $[\![$verscr$(e, e')]\!]$ evaluates to true when the script of the $[\![e']\!]$-th output of the redeeming transaction is equal to $[\![e]\!]$, otherwise it returns $false$.
- $[\![$verrec$(e)]\!]$ evaluates to true when the $[\![e]\!]$-th output of the redeeming transaction is equal to the output that is being redeemed, otherwise it returns $false$.

Until now we have discussed a lot about a transaction's input "redeeming" a certain output, without giving a proper definition. Now that we have a semantics of script, we can be more precise.

Definition 10 (Redeeming an output)**.** We say that the k-th input of a transaction T published at time t can redeem the k'-th output of T' published at time t', and we write $(\mathsf{T}', k', t') \rightsquigarrow (\mathsf{T}, k, t)$, if the following conditions are verified:

1) $\mathsf{T}.\mathsf{in}(k) = (\mathsf{T}', k')$.
2) $[\![\mathsf{T}'.\mathsf{out}(k').\mathsf{scr}]\!]_{(\mathsf{T},k)} = true$.
3) $t \geq \mathsf{T}.\mathsf{absLock}$
4) $t - t' \geq \mathsf{T}.\mathsf{relLock}(i)$.

Finally, we may define the structure that keeps track of all the transactions: the blockchain.

Definition 11 (Blockchain)**.** A *blockchain* **B** is a sequence of pairs (T_i, t_i), such that the sequence of t_i is nondecreasing. If (T, t) is an element of **B** we say that the transaction T *appears at time* t *in* **B**. A blockchain is said to be *consistent* if the following conditions hold:

1) The first pair in **B** is $(\mathsf{T}_0, 0)$ with T_0 initial, and this is the only transaction appearing in the **B**.

2) If T appears in the blockchain at time t, and it is not the first transaction, then each of its inputs redeems an output of some transaction T_j, appearing in $\mathbf{B}$ at an earlier time.
3) Every output of a transaction in $\mathbf{B}$ is referenced at most once by an input of a later transaction.
4) If T is in $\mathbf{B}$, and o_j denotes the output referenced by $\mathsf{T}.\mathsf{in}(j)$, then we have

$$\sum_i \mathsf{T}.\mathsf{out}(i).\mathsf{val} \leq \sum_j \mathsf{o}_\mathsf{j}.\mathsf{val}$$

Meaning that the sum of T outputs' values must not exceed the sum of its inputs' values.

If $\mathbf{B}(\mathsf{T}, t)$ is consistent then we say that (T, t) is a *consistent update* to $\mathbf{B}$. An output of a transaction appearing in $\mathbf{B}$ is said to be *spent* if in the blockchain there appears a transaction that redeems it. The set of unspent outputs is denoted by $UTXO(\mathbf{B})$, (and in general we will abbreviate the expression "unspent transaction output" with UTXO).

Notice how, in this model, a consistent blockchain presents only one initial transaction, meaning that mining is not included in our computational model. We conclude this section by defining deposit outputs: these are outputs with a certain structure, and they will be useful to represent symbolic deposits in the computational model.

Definition 12 (Deposit output)**.** An output o is said to be a *deposit output owned by* A when it has exactly three arguments, the following script

$$\mathsf{scr} = \mathsf{versig}(\mathbf{key}(\mathsf{ctxo}.\mathsf{arg}_3), \mathsf{rtxw}.1),$$

and the third argument is equal to A.

C. The ILLUM compiler

In this appendix we give the full definition of the compiler, which is will allow us to encode symbolic contracts as transaction outputs in the computational model. The compiler will be formalized as a function that takes an initial or continuation advertisement Φ and constructs a transaction T_Φ. First we will focus on the construction of the output(s) of T_Φ; then we will spend a few words in order to describe the various auxiliary inputs taken by the compiler, which are used to construct the other fields of T_Φ. We will then conclude with the full definition of the compiler $\mathbb{B}_{\mathbf{adv}}$.

Constructing the outputs (general). As we mentioned, we will be encoding contracts as transaction outputs. Moreover, in our implementation, all contracts descending from the same initial advertisement will share a common script. We will be able to discern what specific contract is being encoded in an output by looking at its arguments arg, which the script will access in order to enforce the correct execution path. If Φ is initial, then T_Φ will have a single output that represents the newly activated contract; otherwise, if Φ is a continuation advertisement $\left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)\right]_h$, then T will have multiple outputs, either representing deposits (if $\bar{D}$ ends with a `send`), or contracts (if $\bar{D}$ ends with a `call`).

Constructing the outputs (value). The value of each output of T_Φ is easily determined. If $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h$ is a valid initial advertisement, with $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle \equiv \{v\}\ C$, then the single output of T_Φ will have value v. If $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)\right]_h$ is a valid continuation advertisement, with $\bar{D}$ ending in $\mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a_n}\,;\boldsymbol{b_n}\rangle)$, and for each $i = 1 \cdots n$ we have $\mathtt{X}_i\langle \boldsymbol{a_i}\,;\boldsymbol{b_i}\rangle \equiv \{v_i\}\ C_i$, then T_Φ will have n outputs, and the i-th output will have value v_i. Lastly, if Φ is a valid continuation advertisement, with $\bar{D}$ ending in $\mathtt{send}\ (v_1 \to \mathsf{A}_1, \cdots v_n \to \mathsf{A}_n)$, then T_Φ will have n outputs, and the i-th output will have value v_i.

Constructing the outputs (arguments). The value of the arg field of T_Φ will be determined differently depending on the type of advertisement Φ. Here, we also present the various names that we will give to arguments to avoid having to refer to them only by their position in the sequence.

If T_Φ is compiled from an initial advertisement $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h$, with then the first three element of its arg field will be called nonce, branch and name. The first is just a computational counterpart to the symbolic nonce h that appears in Φ, and it gives us a way to force two otherwise identical transactions to be distinct; the second is just a dummy argument, used to make this case more similar to the continuation case, and we will set it to 0; the third is equal to $\mathtt{X}$, the name of the clause that this output will encode.

After those, there are two sequences of arguments α_i and β_i, one for each parameter of the clause $\mathtt{X}(\boldsymbol{\alpha}\,;\boldsymbol{\beta})$: they store the values a_i, b_i specified in Φ.

If instead the transaction is compiled from a continuation advertisement $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)\right]_h$, with $\bar{D}$ ending in $\mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a^1}\,;\boldsymbol{b^1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a^n}\,;\boldsymbol{b^n}\rangle)$, then we need to specify the sequence of arguments for each of its n outputs. For each $k \in \{1, \cdots n\}$, the first three elements of $\mathsf{T}_\Phi.\mathsf{out}(k).\mathsf{arg}$ are again denoted with nonce, branch and name. nonce will again be the counterpart of h; branch will be set to j, which denotes the branch of the "parent" contract that this transaction is continuing; and name will be $\mathtt{X}_k$. Then, we have the α_i and β_i arguments, referring to the parameters of the clause $\mathtt{X}_k$. These will take the values a_i^k and b_i^k specified in $\bar{D}$.

Lastly, if a transaction is compiled from a continuation advertisement $\left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)\right]_h$, with $\bar{D}$ ending in $\mathtt{send}\ (v_1 \to \mathsf{A}_1, \cdots, v_n \to \mathsf{A}_n)$, then each of its n outputs will only need three arguments: nonce, branch, and owner. The value of the first two is set like in the previous case, while the owner argument of the k-th output is set to A_k.

Notation for arguments and expressions. In the construction of the script we will need to replace the parameters α_i, β_i appearing in the contract expressions $\mathcal{E}$ with the respective arguments $\mathsf{ctxo}.\alpha_\mathsf{i}$, $\mathsf{ctxo}.\beta_\mathsf{j}$. In order to make the script more readable we denote this substitution with $\mathsf{ctxo}.\mathcal{E}$. For example if the contract has a term $\mathtt{after}\ t : \cdots$ with $t = \alpha_1 + \alpha_2 + \beta_1 + 3$, we will write $\mathsf{ctxo}.t$ instead of $\mathsf{ctxo}.\alpha_1 + \mathsf{ctxo}.\alpha_2 + \mathsf{ctxo}.\beta_1 + 3$. Similarly, whenever an expression uses the arguments of a redeeming transaction's output, we denote it as $\mathsf{rtxo}(j).\mathcal{E}$. This is less frequent, but needed when dealing with the preconditions of a called clause.

Constructing the outputs (script). The construction of the script is fully detailed in Section 4 of the main text.

Inputs of the compiler. What we have shown until now is how the outputs are constructed starting from a given advertisement term Φ. However, the compiler that we are going to define does not only create the outputs, but an entire transaction, which is the computational counterpart of the symbolic advertisement. In order to do so, the compiler will need to take some additional inputs. First, we take two auxiliary parameters that give us information about the state of the relationship between the symbolic and the computational model, and help us check that the transaction respects certain constraints. These are $\mathbf{key}$, which maps symbolic participants to their public key; and $\mathbf{txout}$, which maps names of contracts or deposits in the current symbolic configuration to outputs (T, j) in the blockchain. Moreover, we take four additional parameters that will be used to construct certain fields of the transaction. These are $\mathbf{in}$, which is a non empty list of outputs (T, j) that will be used to construct the inputs of the transaction; t_0, which is an integer used to construct the absolute timelock; $\mathbf{t}$, which is a list of integers with the same length of $\mathbf{in}$ that will be used to construct the relative timelocks; and $\mathbf{nonce}$, a non empty list of integers that will be used to construct the nonce argument in each output.

Definition 13 (Compiler)**.** Below we define the function $\mathbb{B}_{\mathbf{adv}}$, also known as *compiler*. This definition is structured in two phases: first we show how to construct a transaction T that is the "candidate output" of

$$\mathbb{B}_{\mathbf{adv}}(\Phi, \mathbf{txout}, \mathbf{key}, \mathbf{in}, \mathrm{t}_0, \mathbf{t}, \mathbf{nonce})$$

and then we check that it satisfies certain constraints. If it does then T is the actual output, otherwise the compilation fails and we output $\bot$.

Construction.

- (Inputs) The k-th input of T is set to be equal to $\mathbf{in_k} = (\mathsf{T}_k, j_k)$.
- (Timelocks) The absolute timelock of T is set to be equal to t_0, while the k-th relative timelock is set to be equal to $\mathbf{t_k}$.
- (Output - initial) If $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;w]_h$, then T will only have one output. The output will have $3 + |\boldsymbol{a}| + |\boldsymbol{b}|$ arguments: we have $\mathsf{name} = \mathtt{X}$, $\alpha_\mathsf{i} = a_i$, $\beta_\mathsf{i} = b_i$, and the nonce argument is given by $\mathbf{nonce_1}$.The output's script is constructed as detailed in the above paragraphs, and the output's value is v, where $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle = \{v\}\ C$.
- (Outputs - call) If $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h$, where $\bar{D}$ ends in $\mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a_n}\,;\boldsymbol{b_n}\rangle)$, with $\mathtt{X}_k\langle \boldsymbol{a_k}\,;\boldsymbol{b_k}\rangle \equiv \{v_k\}\ C_k$, then T_Φ has n outputs. Let $\mathbf{in_1} = (\mathsf{T}_1, j_1)$: each of the n outputs of T will have the same script as the j_1-th output of T_1. Moreover, for each k, the k-th output of T will have arguments $\mathsf{nonce} = \mathbf{nonce_k}$, $\mathsf{name} = \mathtt{X}_k$, $\mathsf{branch} = j$, $\alpha_\mathsf{i} = a_i^k$, $\beta_\mathsf{i} = b_i^k$, and value v_k.
- (Outputs - send) If $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x\,,j)\right]_h$, where $\bar{D}$ ends with $\mathtt{send}\ (v_1 \to \mathsf{A}_1, \cdots, v_n \to \mathsf{A}_n)$, then T will have n outputs. For each k, the k-th output will be a deposit output of value v_k owned by A_k in the sense of Definition 12. Its nonce argument will be equal to $\mathbf{nonce_k}$, and branch will be set to j.

Conditions. If one of the following is not satisfied then the return value of $\mathbb{B}_{\mathbf{adv}}$ is set to be $\bot$, otherwise it is T:

- If Φ is a continuation advertisement, then the first input of T must be $\mathbf{txout}(x)$. Aside from that, regardless of the type of advertisement, if Φ takes as input the deposits $\boldsymbol{z}$ then among the inputs of T there must appear (in order) the outputs $\mathbf{txout}(z_j)$, for all $j = 1 \cdots |\boldsymbol{z}|$. If $w = \star$ then T has no other inputs. Otherwise, removing the inputs listed above leaves a list of inputs without a symbolic counterpart (i.e. not in $\operatorname{ran} txout$) such that the amount of their values is equal to w, the sum that Φ takes from the destroyed funds counter.
- If Φ is a continuation advertisement, then $\mathsf{T}.\mathsf{absLock}$ must be greater than all t appearing in any $\mathtt{after}$ decoration in $\bar{D}$. Similarly $\mathsf{T}.\mathsf{relLock}(1)$ must be greater than all δ appearing in any $\mathtt{afterRel}$ decoration.

It's important to notice that the compiler $\mathbb{B}_{\mathbf{adv}}$ does not constructs a new transaction starting from an active contract, but from that the advertisement that propose it. This means that two active contracts that are equal in the symbolic model, may correspond to transactions with very different outputs. We can say that a transaction will remember the whole "history" of a contract, while a symbolic configuration only sees active contracts in the "present". In order to be able to talk, about the actual contract that is being represented in a transaction output, we give the following definition.

Definition 14 (Contract encoded by an output)**.** Take an output o of a compiler generated transaction and a clause $\mathtt{X}$ such that $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle \equiv \{v\}\,C$, and let k_α, k_β be equal to the lengths of $\boldsymbol{a}$, $\boldsymbol{b}$ respectively. We say that the output o *encodes the contract* $\langle C, v\rangle$, if it has a total of $3+k_\alpha+k_\beta$ arguments, and the following equivalences hold:

(i) $\mathsf{o}.\mathsf{arg}_3 = \mathtt{X}$;
(ii) $\forall i \in \{1 \cdots k_\alpha\}.\ \mathsf{o}.\mathsf{arg}_{3+\mathsf{i}} = a_i$;
(iii) $\forall i \in \{1 \cdots k_\beta\}.\ \mathsf{o}.\mathsf{arg}_{3+\mathsf{k}_\alpha+\mathsf{i}} = b_i$;

Notice, that just like deposit outputs, it may be that an output "encoding a contract" does not actually correspond to any active contract in the configuration. However, we will show that if the computational run is coherent to the symbolic run, then at every active contract in the configuration will correspond a unique contract output in the blockchain.

Destroyed funds. At this point, we have talked about how contracts and deposits are represented in the computational model, but there is a third term used by the symbolic model to store currency: the destroyed funds counter $\mathcal{D}(w)$, which must be handled in a different way. There will not be a precise correspondence between $\mathcal{D}(w)$ and some specific outputs: the value w will instead be an approximate representation of every output that does not encode a deposit nor a contract[4].

4. More precisely, this is an approximation from above, as in the next section will prove that the amount of currency stored in outputs that do not encode deposits or contracts is lower or equal to the value w specified by the counter.

The $\star$ symbol. Now that the concept of destroyed funds has been clarified, we can address the difference between choosing $w = \star$ and $w = 0$ in an advertisement term, and show how the computational model justifies using these two different terms in the symbolic setting.

By looking at the compiler's definition (more precisely at the first condition that the constructed transaction needs to satisfy), we can see that if Φ has $w = \star$, then all the inputs of T_Φ will belong to the image of $\mathbf{txout}$, meaning that they all correspond to some symbolic structure (deposits in general, and a contract if Φ is a continuation advertisement). Instead if Φ has $w \neq \star$ then there is at least one input that does not belong to $\mathrm{ran}\,\mathbf{txout}$. In particular, if $w = 0$ this means that those additional inputs have all value 0. This difference can be used to justify the fact that if the symbolic strategy of an honest participants may only choose an action that produces or consumes a symbolic advertisement Φ if the term w inside of Φ is equal to $\star$ (see the third condition of Definition 16).

This requirement is tied to the assumption that whenever a honest participant proposes an action, they want to be sure that the currency it is funded with can actually be spent. In this regard, deposit outputs do not pose any problem: from the structure of the script a participant knows that the output only needs the owner's signature in order to be spent. Instead, outputs that do not have a symbolic counterpart may have an irredeemable script, that prevents them from being spent. In the computational model it might actually be very difficult to confirm if that is indeed the case, but in the symbolic context it is outright impossible. So, we simply assume that honest participants will ignore these funds when proposing a contract.

D. Adversary model

We will now formalize the adversary model for Illum, defining symbolic runs, strategies and conformance. We denote with $\mathsf{Hon} \subseteq \mathsf{Part}$ the set of honest participants, and with $\mathsf{Adv} \notin \mathsf{Part}$ the *symbolic adversary*. We will assume that the adversary is able to control the choices of all dishonest participants.

Randomness in symbolic strategies. As we will soon see, in we will model the choices of participants as probabilistic algorithms, which we construct by giving as considering a deterministic algorithm that takes a random sequence of bits as an additional input. When defining strategies, we will assume that each honest participant A will have access to their own seed r_{A}, and that the adversary has access to r_{Adv}. We define a function r called *randomness source* that associates to each participant (and to the adversary) their random seed. Once the randomness source is assigned, every probabilistic algorithm can be seen as a deterministic one. With a slight notational abuse, the random seed will be listed among the algorithms' inputs only when we explicitly need it.

Definition 15 (Symbolic run)**.** A *symbolic run* R^s is a (possibly infinite) sequence of configurations Γ_i connected by transition labels α_i. The first configuration in the sequence is called *initial configuration* and it is $\Gamma_0 = \Delta_{dep} \mid t_0 \mid \mathcal{D}(0)$, where Δ_{dep} only contains deposits and $t_0 = 0$. If R^s is finite we denote with Γ_{R^s} its last configuration. A run is written as $R^s = \Gamma_0 \xrightarrow{\alpha_0} \Gamma_1 \xrightarrow{\alpha_1} \cdots$. Without loss of generality, we will assume that if an action α_i introduces a new symbolic name, then that name is never used: not only in Γ_i (as the semantics requires), but also in all the other previous configurations.

Definition 16 (Symbolic strategies)**.** A *symbolic strategy* Σ^s_{A} is a probabilistic polynomial time algorithm that takes as input the symbolic run and outputs a finite set of transition labels α, representing the actions that A wants to perform in order to advance the run. The output of Σ^s_{A} (i.e. the set of A's choices) is subject to the following constraints:

1) A must chose labels that are enabled by the semantics. Formally this is expressed by saying that if $\alpha \in \Sigma^s_{\mathsf{A}}(R^s)$, then it exists Γ such that $\Gamma_{R^s} \xrightarrow{\alpha} \Gamma$.
2) A can not impersonate a different participant B and forge their authorizations. In order to express this formally, we first define $\mathcal{A}_{\mathsf{B}}$, the set of labels that express authorizations given by participant B, as

$$\begin{aligned}\mathcal{A}_{\mathsf{B}} = \{\ & auth - in(\mathsf{B},\cdot,\cdot),\ auth - act(\mathsf{B},\cdot,\cdot),\\ & auth - join(\mathsf{B},\cdot,\cdot,\cdot),\ auth - divide(\mathsf{B},\cdot,\cdot,\cdot),\\ & auth - donate(\mathsf{B},\cdot,\cdot)\ \}.\end{aligned}$$

   Then, we have that if $\alpha \in \Sigma^s_{\mathsf{A}}(R^s) \cap \mathcal{A}_{\mathsf{B}}$, then $\mathsf{B} = \mathsf{A}$.
3) A can only choose $adv(\Phi)$ if the term w inside Φ is equal to $\star$. Similarly, A can only choose an $init$, $call$,$send$, or $destroy$ action only if the consumed term Φ has $w = \star$.
4) The strategy is *persistent*, meaning that if at a certain point A chooses an action α (that is not a delay), then at the next step of the run that same action α must be chosen again, if it is still enabled. Formally we can say that if $\alpha \in \Sigma^s_{\mathsf{A}}(R^s)$, $\alpha \neq delay(\delta)$, $R^s \xrightarrow{\alpha'} \dot{R}^s$, and it exists Γ such that $\Gamma_{\dot{R}^s} \xrightarrow{\alpha} \Gamma$; then $\alpha \in \Sigma^s_{\mathsf{A}}(\dot{R}^s)$

Definition 17 (Symbolic adversarial strategy)**.** An adversarial strategy Σ^s_{Adv} is a probabilistic polynomial algorithm that takes as input the run R^s, together with the set of transition labels Λ^s_i chosen by each honest participant A_i. The strategy returns as output a single label λ^s that will be used to update the symbolic run. If $\lambda^s = \Sigma^s_{\mathsf{Adv}}(R^s, \boldsymbol{\Lambda^s})$, then one of the following cases holds:

1) λ^s is neither an authorization nor a delay, and it is enabled by the semantics. Formally we can say that $\lambda^s = \alpha$ where: (i) $\alpha \neq delay(\delta)$; (ii) there is no B for which $\alpha \in \mathcal{A}_{\mathsf{B}}$; (iii) there exists Γ such that $\Gamma_{R^s} \xrightarrow{\alpha} \Gamma$.
2) λ^s is an authorization given by a honest participant, and it is chosen by the strategy of the corresponding participant. In short we can say that if $\lambda^s \in \mathcal{A}_{\mathsf{A}_i}$ then we also must have $\lambda^s \in \Lambda^s_i$.
3) λ^s is an authorization given by a dishonest participant $\mathsf{B} \notin \mathsf{Hon} = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$ and it is enabled by the semantics.
4) λ^s is a delay and it is chosen by the strategies of all honest participants. Symbolically this is $\lambda^s = delay(\delta)$ and $\forall i \in \{1 \ldots k\}$. ($\Lambda^s_i = \emptyset$ or $delay(\delta_i) \in \Lambda^s_i$ with $\delta_i \geq \delta$)

Definition 18 (Symbolic conformance)**.** Given a random source r and a set of strategies $\mathbf{\Sigma}^s$ that includes those of honest participants $\mathsf{A}_1, \ldots, \mathsf{A}_k$ and the adversarial strategy Σ^s_{Adv}, it is possible to uniquely determine a run R^s. We say that this run *conforms* to the pair $(\mathbf{\Sigma}^s, r)$. More precisely, the conformance relations between R^s and $(\mathbf{\Sigma}^s, r)$ holds if and only if one of the two following conditions is verified

1) $R^s = \Gamma_0$ where $\Gamma_0 = (\parallel_j \langle \mathsf{A}_j, v_j \rangle_{x_j}) \mid 0 \mid \mathcal{D}(0)$ is an initial configuration.
2) $\dot{R}^s \xrightarrow{\alpha} R^s$ where $\dot{R}^s$ conforms to $(\mathbf{\Sigma}^s, r)$ and, given $\forall i.\ \Lambda^s_i = \Sigma^s_{\mathsf{A}_i}(\dot{R}^s, r_{\mathsf{A}_i})$, we have $\Sigma^s_{\mathsf{Adv}}(\dot{R}^s, \mathbf{\Lambda^s}, r_{\mathsf{Adv}}) = \alpha$

If we are considering a set $\mathbf{\Sigma}^s$ that does not include an adversarial strategy, then we say that a run R^s conforms to $(\mathbf{\Sigma}^s, r)$ if there exists an adversarial strategy Σ^s_{Adv} such that R^s conforms to $(\{\Sigma^s_{\mathsf{Adv}}\} \cup \mathbf{\Sigma}^s, r)$.

Computational adversary model. Below, we model adversaries at the computational level. We will follow a structure similar to what we did above for the symbolic model.

Randomness and keys. The choices of participants will again be modelled as probabilistic algorithms. For this reason, we provide a randomness source r_{A} to each honest participant, and one to the adversary. In the computational model, these randomness sources are not only passed as an additional input to the strategies in order to make then a probabilistic, but they are also used to generate the participants' keys. Given a security parameter η, we have that each honest participant will use the first η bits of their random sequence to produce an asymmetric key pair, $K_{\mathsf{A}}(r_{\mathsf{A}})$. This key will be used to produce a witness by signing a transaction, whenever the script requires it. The public and secret part of the key K are denoted respectively with K^p and K^s. Similarly, the adversary will use their random source r_{Adv} to generate the keys of all dishonest participant, using η bits for each key pair. With a slight notational abuse, the random seed will not be listed among the inputs of the algorithms that use it, unless it is explicitly needed.

Definition 19 (Computational run)**.** A *computational run* R^c is a (possibly infinite) sequence of labels λ^c, each encoding one of these possible actions.

T	appending transaction T to the blockchain
δ	performing a delay
$\mathsf{A} \to * : m$	broadcasting of message m from A

Every computational run starts with the initial transaction T_0 that distributes a certain quantity of currency to each participant. We will assume that all of T_0's outputs are deposit outputs. Immediately after that, every participant broadcasts the public part of the key that they have generated using their random source. So we have

$$R^c_0 = \mathsf{T}_0 \cdots \mathsf{A}_i \to * : (K^p_{\mathsf{A}_i}(r_{\mathsf{A}_i})) \cdots \\ \cdots \mathsf{B}_j \to * : (K^p_{\mathsf{B}_j}(r_{\mathsf{Adv}})) \cdots$$

where all outputs of T_0 are standard, A_i are all the participants in Hon, and B_j are all the others.

Definition 20 (Blockchain of computational runs)**.** In order to keep track of the transactions that occur in R^c, while ignoring the messages, we define the *blockchain of the computational run* $\mathbf{B}_{R^c}$ as follows:

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

where δ_{R^c} denotes the sum of the delays present in the run up until that point.

We can use the notion of consistency for a blockchain, to define consistency for the runs.

Definition 21 (Consistent computational run)**.** A computational run R^c is said to be *consistent* if:

1) Its blockchain $\mathbf{B}_{R^c}$ is consistent.
2) If $R^c = \dot{R}^c\, \mathsf{T} \cdots$, then among the labels of $\dot{R}^c$ we can find, in this order: a message $\mathsf{B} \to * : \mathsf{T}$ that encodes the transaction sent after all T's timelocks have expired[5]; and all messages $\mathsf{B} \to * : (\mathsf{T}, j, wit, i)$ where wit is the i-th witness of the j-th input of T. These messages may be sent from different participants.

The second condition required to have a consistent run amounts to saying that a transaction and its witnesses are broadcast before they are appended to the blockchain. This is a reasonable assumption: the blockchain is public, so in order to include a transaction one also has to broadcast it. If $R^c \lambda^c$ is a consistent run, then we say that λ^c is a *consistent update* to R^c. Note that delays and messages are always consistent updates to a run. We now present the *computational strategies*, the way in which participants can interact with the run, updating it.

Definition 22 (Computational strategies)**.** A *computational strategy* for an honest participant A is a probabilistic polynomial algorithm Σ^c_{A}, that takes as input the computational run R^c and outputs a finite set of computational labels Λ^c, whose elements λ^c consistently update R^c. Moreover, if λ^c is a message, then it is sent from A. We require strategies to be persistent: if $\lambda^c \in \Sigma^c_{\mathsf{A}}(R^c)$ is not a delay, and $\dot{\lambda}^c \neq \lambda^c$ is such that both $R^c \dot{\lambda}^c$ and $R^c \dot{\lambda}^c \lambda^c$ are consistent, then we must have $\lambda^c \in \Sigma^c_{\mathsf{A}}(R^c \dot{\lambda}^c)$.

Definition 23 (Computational adversarial strategies)**.** The *computational adversarial strategy* is a polynomial algorithm Σ^c_{Adv} that takes as input the computational run and the set of choices taken by each honest participant, and returns as output a single computational label used to update the run. If $\lambda^c = \Sigma^c_{\mathsf{Adv}}(R^c, \mathbf{\Lambda^c})$ then λ^c is a consistent update to R^c, with the following constraint: if $\lambda^c = \delta$ is a delay, then it must have been chosen by all the honest participants' strategies. Formally, we can express this as $\forall i \in \{1, \ldots, k\}.\ (\Lambda^c_i = \emptyset$ or $\delta_i \in \Lambda^c_i$ with $\delta_i \geq \delta)$.

Notice that we are allowing the adversary to impersonate every participant B, by introducing in the run messages

5. In order to formalize what is meant by "all T's timelocks have expired", assume the following: (i) the j-th input of T is an output of a transaction T_j which appears in $\mathbf{B}_{R^c}$ at time t_j; (ii) the sum of delays from the beginning of the run up until the sending of the message is t ; (iii) the transaction has relative timelocks $\mathsf{T}.\mathsf{relLock}(j) : \delta_j$ and an absolute timelock $\mathsf{T}.\mathsf{absLock} : t_0$. We say that all T's timelocks have expired if $t \geq t_0$ and $\forall j.\ t - t_j \geq \delta_j$.

$\mathsf{B} \to * : m$. However, since they do not have direct access to B's random source, they will be, with overwhelming probability, unable to forge B's signature, since they are restricted to using a polynomial time strategy.

Definition 24 (Computational conformance)**.** Take a randomness source r, and a set of strategies $\mathbf{\Sigma}^c$ containing those of all honest participants $\{\mathsf{A}_1, \cdots, \mathsf{A}_k\}$ as well as the adversarial strategy. We say that a run R^c *conforms* to $(\mathbf{\Sigma}^c, r)$ if one of the two following conditions holds

1) R^c is initial, with keys derived from the randomness source r.
2) $R^c = \dot{R}^c \lambda^c$ with $\dot{R}^c$ conforming to $(\mathbf{\Sigma}^c, r)$, and, given $\Lambda^c_i = \Sigma^c_{\mathsf{A}_i}(\dot{R}^c, r_{\mathsf{A}_i})$, we have $\Sigma^c_{\mathsf{Adv}}(\dot{R}^c, \mathbf{\Lambda^c}, r_{\mathsf{Adv}}) = \lambda^c$.

E. Coherence

In this appendix we formally define the *coherence* relation between symbolic and computational runs, and we prove some of its properties.

Before we begin with the definition, it is important to note that the coherence relation does not only involve the two runs, but also three auxiliary functions, $txout$, key, and $prevTx$. The first two are essentially the same functions that we used in the previous chapter to provide context to the compiler, and they are used to map symbolic names to transaction outputs; to map participants to their public deposit's key. The third function $prevTx$ maps a symbolic advertisement term Φ to a transaction T_Φ that encodes it. It will be used to keep track of previous advertisements to avoid repeating them.

Definition 25 (Coherence)**.** The definition of the coherence relation

$$coher(R^s, R^c, txout, key, prevTx)$$

is split into three sections, each one consisting of one or more inductive cases.

Base case. The relation:

$$coher(R^s, R^c, txout, key, prevTx)$$

holds if the following conditions are verified:

1) $R^s = \Gamma_0$ it is an initial configuration;
2) key maps each owner of a deposit in R^c to a public key;
3) R^c is initial, and the keys broadcast after the first transaction are the ones in the image of key;
4) $txout$ maps exactly the name x of each deposit $\langle \mathsf{A}, v\rangle_x$ in Γ_0 to a different deposit output of T_0 of value v owned by A[6];
5) $prevTx$ is the empty function.

6. Note that we may not have a unique way to determine $txout$: for example this happens if in the initial configuration a participant owns two deposits of the same value. We do not give a detailed explanation on how to handle this edge cases: the key fact here is that even in those case it is still possible to construct $txout$ so that it is injective.

Inductive case 1. The relation:

$$coher(R^s \xrightarrow{\alpha} \Gamma, R^c \lambda^c, txout, key, prevTx)$$

holds if $coher(R^s, R^c, txout', key', prevTx')$ holds, in addition to one of the following conditions.

In all cases where explicit changes are not mentioned we will assume that $txout = txout'$, $key = key'$, and $prevTx = prevTx'$.

1) $\alpha = msg(\Theta)$, $\lambda^c = \mathsf{A} \to * : \mathcal{C}$, where Θ is an incomplete advertisement and $\mathcal{C}$ is a bitstring that encodes it. In $\mathcal{C}$ every deposit name z_j is represented by the transaction output $txout(z_j)$.
2) $\alpha = adv(\Phi)$, $\lambda^c = \mathsf{A} \to * : \mathsf{T}_\Phi$, where $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\,;\boldsymbol{z}\,;\,w]_h$ is a valid initial advertisement in R^s and
$$\mathsf{T}_\Phi = \ddot{\mathtt{B}}_{\mathbf{adv}}(\Phi, txout', key', \mathbf{in}, \mathrm{t}_0, \mathbf{t}, \mathbf{nonce})$$
for some $\mathbf{in}$, t_0, $\mathbf{t}$ and $\mathbf{nonce}$. Moreover, we require this to be is the first time that λ^c appears in R^c after all timelocks in T_Φ are expired. The function $prevTx$ extends $prevTx'$ by mapping Φ to T_Φ.
3) $\alpha = adv(\Phi)$, $\lambda^c = \mathsf{A} \to * : \mathsf{T}_\Phi$, where $\Phi = \left[\bar{D}\,;\boldsymbol{z}\,;\,w\,;\,(x\,,j)\right]_h$ is a valid continuation advertisement in Γ_{R^s} and
$$\mathsf{T}_\Phi = \ddot{\mathtt{B}}_{\mathbf{adv}}(\Phi, txout', key', \mathbf{in}, \mathrm{t}_0, \mathbf{t}, \mathbf{nonce})$$
for some $\mathbf{in}$, t_0,$\mathbf{t}$, and $\mathbf{nonce}$. We have the same additional restriction of (2), and again $prevTx$ extends $prevTx'$ by mapping Φ to T_Φ.
4) $\alpha = init(\Phi, x)$, $\lambda^c = \mathsf{T}_\Phi$, where $\mathsf{T}_\Phi = prevTx'(\Phi)$. In the symbolic setting this step produces $\langle C, v\rangle_x$, and so we extend $txout'$ to $txout$ by mapping x to the single output of T_Φ.
5) $\alpha = call(\Phi)$, $\lambda^c = \mathsf{T}_\Phi$, where $\mathsf{T}_\Phi = prevTx'(\Phi)$. With this action the symbolic run produces the active contracts $\langle C_1, v_1\rangle^t_{y_1} \cdots \langle C_k, v_k\rangle^t_{y_k}$, so we extend $txout'$ to $txout$ by mapping each y_i to the i-th output of T_Φ.
6) $\alpha = send(\Phi)$, $\lambda^c = \mathsf{T}_\Phi$, where $\mathsf{T}_\Phi = prevTx'(\Phi)$. This time the symbolic action doesn't produce any contract, but deposits $\langle \mathsf{A}_1, v_1\rangle_{y_1}, \cdots, \langle \mathsf{A}_k, v_k\rangle_{y_k}$. For this reason $txout'$ is extended by mapping $txout(y_i)$ to be equal to the i-th output of T_Φ.
7) $\alpha = auth - action(\mathsf{A}, \Phi)$, $\lambda^c = \mathsf{B} \to * : m$, where m is a quadruple $(\mathsf{T}_\Phi, 1, wit, i)$. We have that wit is a the signature with A's key on the first input of $\mathsf{T}_\Phi = prevTx'(\Phi)$. The index i is the position of wit as witness, and can be deduced by looking at the script of $prevTx'(\Phi)$. Moreover, we want λ^c to be is the first instance of this signature being sent after a broadcast of T_Φ that, in turn, appears after all of T_Φ's timelocks have expired.
8) $\alpha = auth - in(\mathsf{A}, z, \Phi)$, $\lambda^c = \mathsf{B} \to * : (m, i)$, where m is a quadruple $(\mathsf{T}_\Phi, j, wit, 1)$, with being a signature with A's key on the j-th input of the transaction $\mathsf{T}_\Phi = prevTx'(\Phi)$. As in (8), we want this to appear in the run after a message encoding T_Φ (which, again, has been sent after all T_Φ's timelocks have expired). Moreover we ask that the j-th input is exactly $txout'(z)$, or equivalently that z is the j-th deposit specified in Φ. Differently from (8) this case may happen even with advertisements of any form.

9) $\alpha = delay(\delta)$, $\lambda^c = \delta$. Symbolic delays trivially translate to computational ones, without changing the mapping between outputs and names. This means that two coherent runs always have the same time.
10) $\alpha = auth - join(\mathsf{A}, x, x', i)$, $\lambda^c = \mathsf{B} \to * : m$, where m is a quadruple (T, i, wit, j) such that wit signature on input $txout'(x)$ for a previously broadcast transaction T that takes two inputs ($txout'(x)$ and $txout'(x')$, in order) and has a single output that encodes a deposit $\langle \mathsf{A}, v + v' \rangle$. Moreover λ^c is the first instance of such signature being broadcast in the computational run (from the broadcast of T onward).
11) $\alpha = join(x, x')$, $\lambda^c = \mathsf{T}$, where T has exactly two inputs given by $txout'(x)$ and $txout'(x')$, of value v and v' respectively, and a single deposit output owned by A of value $v + v'$. In the symbolic run the action α consumes the two deposits x and x' in order to produce $\langle \mathsf{A}, v + v' \rangle_y$. We extend $txout'$ by mapping $txout(y)$ to the output of T.
12) $\alpha = auth - divide(\mathsf{A}, x, v, v')$, continues like (10).
13) $\alpha = divide(x, v, v')$, continues like (11)
14) $\alpha = auth - donate(\mathsf{A}, x, \mathsf{B})$, continues like (10)
15) $\alpha = donate(x, \mathsf{B})$, continues like (11)
16) $\alpha = adv(\Phi)$, $\lambda^c = \mathsf{A} \to * : \mathsf{T}$, where Φ is a destroy advertisement $[\boldsymbol{z} \,;\, w]_h$; the transaction T is not compatible with either one of the two previous $adv()$ cases, nor it represents a join, divide or donate operation. Moreover, among the inputs of T there appear (in order) the outputs $\mathbf{txout}(z_j)$, for all $j = 1 \cdots |\boldsymbol{z}|$. If $w = \star$ then these are all the inputs, otherwise removing those leaves a list of inputs outside of $\operatorname{ran} txout'$, such that the sum of their values is w. Lastly, we have the same additional restrictions of (2), and $prevTx$ extends $prevTx'$ by mapping Φ to T.
17) $\alpha = destroy(\Phi)$, $\lambda^c = \mathsf{T}$, where $\mathsf{T} = prevTx'(\Phi)$. Notice that from the advertisement of Φ we know that λ^c cannot correspond to any of the already mentioned cases.

Inductive case 2. The predicate:

$$coher(R^s, R^c \lambda^c, txout, key, prevTx)$$

holds if $coher(R^s, R^c, txout, key, prevTx)$ holds, in addition to one of the following conditions:

1) $\lambda^c = \mathsf{T}$ with no input of T belonging to the image of $txout$.
2) $\lambda^c = \mathsf{A} \to * : m$ that does not correspond to any of the symbolic moves described in the first inductive case of the definition.

Now that we have formalized the concept of coherence, we can establish some results about the relation between the two models. Notice that, for most of the following propositions, a rigorous proof by induction would need to examine all 20 inductive rules appearing in the definition. For the sake of brevity we will focus only on the cases that are relevant to each proof. In most cases we will only be interested in the runs and the $txout$ map, so we will write $R^s \sim_{txout} R^c$.

We start with a lemma regarding the $txout$ map.

Lemma 3. *Assuming $coher(R^s, R^c, txout, key, prevTx)$ holds, the map $txout$ is injective.*

Proof. By induction. In the base case we are mapping every deposit to a different output of T_0, so $txout$ is injective. In all of the inductive cases we can assume the injectivity of $txout'$, and we will need to check that $txout$ remains injective. Obviously, we will only need to look at the cases that have $txout \neq txout'$.

For this reason, we are only concerned with cases corresponding to $init$, $call$, $send$, $join$, $divide$, and $donate$ actions. All of these create new symbolic names, which are then mapped to outputs of a newly created transaction that is appended to the blockchain in that very step. This means that all these outputs are different from all the ones in $\operatorname{ran} txout$ (since the transaction they belong to was not present on the blockchain in previous steps). This, in conjunction with the fact that each new name is mapped to a different output of the new transaction, proves that the new map $txout$ is still injective. □

Next we have a proposition that clarifies the relationship between unspent outputs in the computational model's blockchain and active contract or deposit in the symbolic configuration. The proposition will also shows that if coherence holds, then the $txout$ map does what it is intuitively expected to do: it associates deposits and contracts to transaction outputs of the same value.

Proposition 4. *Assume that:*

$$coher(R^s, R^c, txout, key, prevTx)$$

and let x be the name of a deposit or a contract in the symbolic run R^s. If x is the name of a deposit or of an active contract in Γ_{R^s}, then the output $txout(x)$ is unspent in $\mathbf{B}_{R^c}$. If instead x is the name of a deposit or contract in R^s, appearing in some previous configuration but not in Γ_{R^s}, then the output $txout(x)$ is spent $\mathbf{B}_{R^c}$. Moreover, the map $txout$ preserves the value, so that deposits (resp. active contracts) of value (resp. balance) v are always mapped to outputs of value v.

Proof. By looking at the previous proof, we can see that once $txout(x)$ is determined, it does not change. So, we just need to prove two statements: (i) whenever a new contract or deposit is inserted in the configuration, the domain of $txout$ is extended, and the image of that new name is a newly created output (which is obviously unspent) of correct value; (ii) a UTXO belonging to $\operatorname{ran} txout$ is spent in the computational setting if and only if its pre-image is consumed in the symbolic run.

To prove (i) notice that in Definition 25 the cases in which the symbolic action creates a new deposit or a new contract are exactly the cases in which the domain of $txout$ is extended. These all happen in the first set of inductive cases, in the items related to the following operations: $send$, $join$, $divide$ and $donate$ (for deposits), $init$ and $call$ (for contracts). We can immediately see from the coherence definition that all these cases append a transaction T to the computational run. Moreover, this T has the correct number of outputs; and $txout$ is updated accordingly. The fact that the output's value matches the symbolic value is ensured either by a direct specification in the coherence definition (for $join$, $divide$, and $donate$ operations), or by the script's covenant [7] in cases that

7. Later, in Proposition 5 we will give a more profound justification on why the covenant really forces the value to be correct

append a compiler generated transaction (which happens in *send*, *init* and *call* operations).

To prove the second condition we need to only check the inductive steps that spend a UTXO associated to some symbolic name, or the ones that remove some symbolic name from the configuration. By looking at the definition of coherence we can see that these two cases coincide. They happens only in the first set of inductive cases, and specifically in the cases related to *init*, *call*, *send*, *join*, *divide*, *donate* and *destroy* operations. The semantic transition rule for each of those actions consumes some deposit or contract, and we can see (either thanks to a direct specification in the coherence definition, or to the definition of the compiler) that the corresponding transaction appended to the computational run always spends the corresponding UTXOs. □

The next result further refines the above proposition, by showing that we can always determine the structure of an output associated to a symbolic term. It also serves as a justification for the definitions of deposit output and of output encoding a contract that were given in the previous sections.

Proposition 5. *Assume that:*

$$coher(R^s, R^c, txout, key, prevTx)$$

If an output in R^c is the image of a symbolic name, then we can fully determine its structure

- *If $\langle \mathsf{A}, v\rangle_x$ is in R^s, then $txout(x)$ is a deposit output owned by A.*
- *If $\langle C, v\rangle_x^t$ is in R^s, then $txout(x)$ is the output of a compiler generated transaction. Moreover $txout(x)$ is a contract output encoding C.*

Proof. Again, we proceed by induction. In the base case there is no contract in the configuration, and the symbolic deposits are all mapped by $txout$ to deposit outputs of T_0, so both statements hold. Among the inductive cases we only need to check those that introduce a deposit or a contract in the symbolic configuration, and hence extend $txout'$. We start with the deposit operations *join*, *divide*, and *donate*. In those cases, the definition of coherence explicitly states that $txout'$ is extended by mapping the newly created deposit to a deposit output, with the correct value and owner.

This only leaves us with the $send(\Phi)$ case for deposits, and the $init(\Phi)$ and $call(\Phi)$ cases for contracts. In those cases the inductive step adds the transaction $\mathsf{T}_\Phi = prevTx'(\Phi)$ to the computational, and extends the $txout$ relation by mapping the newly created symbolic terms (deposits or contracts) to T_Φ's outputs.

Notice that if Φ is a valid initial or continuation advertisement term in the symbolic run, then the transaction $prevTx'(\Phi)$ must be compiler generated. This fact can easily be proved by induction on the coherence definition: in the base case $prevTx'$ is the empty map, and the only inductive cases that we need to check are the ones that modify $prevTx$, extending its domain to a new initial or continuation advertisement Φ. Those cases are the one corresponding to a symbolic $adv(\Phi)$ operation, and the conditions that they need to satisfy directly imply that $prevTx(\Phi)$ must be compiler generated.

But now, if we look at how the compiler generates the output(s) of T_Φ, we see that if $\Phi = \left[\bar{D}\ ;\ \boldsymbol{z}\ ;\ w\ ;\ (x,j)\right]_h$, with $\bar{D}$ ending in $\texttt{send}\ (\mathsf{A}_1 \to v_1, \cdots \mathsf{A}_n \to v_n)$, then (by the compiler definition) the i-th output of T_Φ will be a deposit output of value v_i owned by A_i, and (by definition of coherence) it will be the image under $txout$ of i-th deposit created by $send(\Phi)$, which proves our claim for this inductive case. If instead $\Phi = \left[\bar{D}\ ;\ \boldsymbol{z}\ ;\ w\ ;\ (x,j)\right]_h$ with $\bar{D}$ ending in $\texttt{call}\ (\mathtt{X}_1\langle \boldsymbol{a^1}\,;\boldsymbol{b^1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a^n}\,;\boldsymbol{b^n}\rangle)$, with $\mathtt{X}_i\langle \boldsymbol{a^i}\,;\boldsymbol{b^i}\rangle \equiv \{v_i\}\ C_i$; then (by the compiler definition) the i-th output of T_Φ will be an output of value v_i encoding the contract C_i, and (by definition of coherence) it will be the image under $txout$ of i-th contract created by $call(\Phi)$, which proves our claim for this inductive case. The initial advertisement case is identical to the $call(\Phi)$ seen above, with only one output. □

The above propositions state that it is possible to "keep track" of symbolic deposits and active contracts by seeing them as computational outputs. However, there is another term in the symbolic configuration that is used to store an amount of currency usable by the participants: the destroyed fund counter $\mathcal{D}$. In the following proposition we will show how the counter approximates from above the amount of currency contained in output that do not correspond to any other symbolic term.

Proposition 6. *Assume that:*

$$coher(R^s, R^c, txout, key, prevTx)$$

The sum of all values stored in transaction outputs that do not belong to $\mathrm{ran}\, txout$ *is smaller or equal to the value w specified by the destroyed fund counter $\mathcal{D}(w)$ in Γ_{R^s}.*

Proof. By induction. In the base case all outputs in R^c are images of symbolic deposits, and Γ_0 contains $\mathcal{D}(0)$, so this proposition holds.

In the definition of coherence there are two sets of inductive cases: in either of them, we will look at the inductive premise and denote with w_0 the amount of funds contained in the destroyed funds counter present in the last configuration of the symbolic run, and with W_0 the sum of the values of all unspent transaction outputs in the computational blockchain that do not correspond to symbolic deposits or contracts. This means that our inductive hypothesis states that $W_0 \leq w_0$, and, after having defined W_1 and w_1 in a similar way, we want to prove that $W_1 \leq w_1$.

While looking at the items in the first set of inductive cases in Definition 25, we only care about proving our proposition in the cases that either modify the counter with a symbolic action, or insert in the computational blockchain a transaction that spends or produces some inputs outside of $\mathrm{ran}\, txout$.

When the symbolic action is $init(\Phi)$, $call(\Phi)$, or $send(\Phi)$, the corresponding transaction T_Φ may spend some inputs that do not have a symbolic correspondent. Since T_Φ must be compiler generated, we know that the total value of these inputs must amount exactly to the value w that appears in Φ (or to 0 if $w = \star$). By spending these outputs we decrease W_0 to $W_1 = W_0 - w$. The semantics of these actions tells us that, in the symbolic run, the amount w is removed from the counter, giving us

$\mathcal{D}(w_1)$ with $w_1 = w_0 - w$. This means that the inequality $W_1 \leq w_1$ holds.

Next, we need to check what happens when the symbolic action is $destroy([\boldsymbol{z}\,;\,w]_h)$. The semantics of $destroy$ tells us that the value w_0 in the counter is increased by $\sum_j u_j$, where each u_j is the value contained in the deposit z_j. In the computational run instead we are creating a transaction T which spends w from inputs without a symbolic counterpart. Let us denote with w' the sum of the values of T's outputs. This value must be smaller or equal to the sum of T's input values, which is $\sum_j u_j + w$. Moreover, notice that none of T's output will have a symbolic counterpart. For this reason, the total the value stored by outputs without a symbolic counterpart becomes $W_1 = W_0 - w + w'$, and since we have $w' \leq \sum_j u_j + w$, this gives us $W_1 \leq W_0 + \sum_j u_j$. In turn, this implies $W_1 \leq w_1 = w_0 + \sum_j u_j$, and the inequality is preserved.

Lastly, we look at the second set of inductive definitions, where the first case tells us that we can insert any transaction T whose input do not have a symbolic counterpart without performing any action on the symbolic run. T may only reduce the total amount of funds stored in outputs that are outside of $\operatorname{ran} txout$, since the sum of the values of its inputs must be greater or equal to the sum of the values of its outputs. This means that $W_1 \leq W_0$ while $w_1 = w_0$, and the inequality holds. □

F. Correctness of the compiler

In our implementation of ILLUM, the logic of contracts is only enforced through the script of a compiler-generated transaction. By looking at how such scripts are constructed, it is intuitively obvious that they can be redeemed only by following the symbolic contract logic: in this section we will prove two theorems that justify more precisely why that is actually true, showing that the compiler correctly implements the language.

From Proposition 5 we know that if the two runs are coherent, each active contract corresponds to a transaction's output that encodes it. However, we want to make sure that the only transactions that are able to redeem an output that encodes an active contract are the compiler generated ones. This is important because otherwise it would be really easy to "break" the coherence relation, by redeeming the balance of a contract with a transaction that is not compiler generated, and hence does not carry any meaning to the symbolic setting. The following theorem proves that this may never happen.

Theorem 7. *Assume that:*

$$coher(R^s, R^c, txout, key, prevTx)$$

and let $\langle C, v\rangle^t_x$ be an active contract in Γ_{R^s}. Take a transaction T that consistently updates R^c, and has an input that redeems the UTXO $txout(x)$. Then T is compiler generated (and possibly completed by including witnesses), meaning that there exists Φ, $\mathbf{in}$, t_0, $\mathbf{t}$, $\mathbf{nonce}$ such that

$$\mathsf{T} = \mathbb{B}_{\mathbf{adv}}(\Phi, txout, key, \mathbf{in}, \mathrm{t}_0, \mathbf{t}, \mathbf{nonce})$$

for some Φ, $\mathbf{in}$, t_0, $\mathbf{t}$, $\mathbf{nonce}$. Moreover, the input of T that redeems $txout(x)$ is the first, and we have $\Phi = \left[\bar{D}\,;\,\boldsymbol{z}\,;\,w\,;\,(x\,,j)\right]_h$, for some values $\boldsymbol{z}$, w, x, h, and for $\bar{D}$, j, such that $\bar{D} \approx D$ where D is the j-th branch of C.

Proof. The proof is organized in two parts: first we will show how to construct the terms Φ, $\mathbf{in}$, t_0, $\mathbf{t}$, $\mathbf{nonce}$ starting from the fields of a transaction T that redeems $txout(x)$; then we will prove that using the constructed terms as inputs for the compiler yields exactly T.

Constructing $\mathbf{in}$ is easy: we just take $\mathbf{in_i}$ to be the i-th the input of T. Obviously, one of these inputs will be $txout(x)$. By Proposition 5 $txout(x)$ is compiler generated, so we know the structure of its script. The first part of the script sets the condition $\mathsf{inidx} = 1$, which tells us that the output must be redeemed by an input in position 1. This means that $\mathbf{in_1} = txout(x)$. The fact that this same instructions is present in every compiler generated transaction means that none of the other inputs of T can be the image of a contract in the symbolic configuration. This implies that all other inputs of T are either outside of $\operatorname{ran} txout$, or are the image of a deposit. We are now able to construct $\boldsymbol{z}$ and w: the first is constructed by taking the preimage of all inputs of T that are in $(\operatorname{ran} txout \setminus \{txout(x)\})$, and the other is set to be the sum of the values of the inputs that are not in $\operatorname{ran} txout$ (or it is set to $\star$ if there are no such inputs). t_0 is set to be equal to T's absolute timelock, while every other $\mathbf{t_i}$ is set to the value of $\mathsf{T}.\mathsf{relLock}(i)$. We construct $\mathbf{nonce_k}$ by taking the first argument of the k-th output of T (this is possible since, as we will show later in the proof, each output of T has more than one argument).

Choosing $\bar{D}$ and j requires a bit more work. Note that in the rest of the proof we will be referring to arguments by their name, instead of more precisely tracking their position. The paragraph "Constructing the outputs: arguments" of the previous appendix motivates why we are able to do so.

From Proposition 5 we know that $txout(x)$ encodes contract C, which means that T will have to satisfy scr_C. This term is organized as a conditional check, so T must satisfy one of its branches. We assume that the taken branch is the k-th:

$$\mathsf{if}\ \mathsf{B_k}\ \mathsf{then}\ \mathsf{scr}_{D_k},$$

where $\mathsf{B_k}$ is a shorthand for the expression

$$\mathsf{outlen(rtx)} = n_k\ \mathsf{and}\ \mathsf{rtxo}(1).\mathsf{branch} = k\ \mathsf{and}$$
$$\cdots\ \mathsf{and}\ \mathsf{rtxo}(n_k).\mathsf{branch} = k.$$

The value of branch argument (the second) will then be the same across all outputs of T. The value j, which represent the branch in the symbolic advertisement, will be set to be equal to the second argument of any output of T.

We can now use the fact that T must satisfy scr_{D_k} in order to construct the last term, $\bar{D}$. We have two cases, since D_k ends either in a `send` or in a `call`. Now that we know which is the branch that is being executed, we can easily inspect C, to determine which of these two cases we are dealing with. If D_k ends in a `send`, then we set $\bar{D}$ to be equal to D_k. If instead D_k ends in `call` $(\mathtt{X}_1\langle \boldsymbol{a^1}\,;\mathtt{?^1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a^n}\,;\mathtt{?^n}\rangle)$, then, in order to construct $\bar{D}$, we need to "complete" it, assigning a value to the placeholders. The script scr_{D_k} specifies that the i-th output has $3+|\boldsymbol{\alpha^i}|+|\boldsymbol{\beta^i}|$ arguments, where α^i_l and β^i_h are

the parameters in $\mathtt{X}_i$, the i-th called clause. For this reason we are able to construct $\bar{D}$ by filling the placeholders $\mathbf{?^i}$ with the last $|\boldsymbol{\beta^i}|$ arguments of the i-th output of T.

At this point we have constructed the terms Φ, $\mathbf{in}$, t_0, $\mathbf{t}$, $\mathbf{nonce}$, so we can pass them as inputs to the compiler and construct

$$\mathsf{T}' = \ddot{\mathbb{B}}_{\mathbf{adv}}(\Phi, txout, key, \mathbf{in}, \mathrm{t}_0, \mathbf{t}, \mathbf{nonce})$$

It is easy to check that the conditions set by the compiler are satisfied, proving that T' is a proper transaction and not $\bot$.

1) We already know that $\mathbf{in_1}$ is $txout(x)$. The rest of the condition follows from the fact that $\mathbf{in}$, $\boldsymbol{z}$ and w have been constructed together, starting from T's inputs.
2) We know that t_0 and $\mathbf{t_1}$ have been constructed from T's timelocks. But these timelocks must be greater than the value appearing in the `after` (and respectively `afterRel`) decorations of $\bar{D}$, since the $txout(x)$'s script specifies the conditions

$$\mathsf{scr}_{\mathtt{after}\, t\, :\, D} = \mathsf{absAfter}\ \mathsf{ctxo}.t : \mathsf{scr}_D$$
$$\mathsf{scr}_{\mathtt{afterRel}\, \delta\, :\, D} = \mathsf{relAfter}\ \mathsf{ctxo}.\delta : \mathsf{scr}_D.$$

In order to conclude the proof, we now need to show that $\mathsf{T} = \mathsf{T}'$.

1) (Inputs and timelocks)
   The inputs and timelocks of T' are determined by $\mathbf{in}$, t_0 and $\mathbf{t}$. By constructions of the parameters, T' must have the same inputs and timelocks of T.
2) (Outputs - call)
   If $\bar{D}$ ends in $\mathtt{call}\ \mathtt{X}_1\langle \boldsymbol{a^1}; \boldsymbol{b^1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a^n}; \boldsymbol{b^n}\rangle$, with $\mathtt{X}_i\langle \boldsymbol{a^i}; \boldsymbol{b^i}\rangle \equiv \{v_i\}\ C_i$, then we have the following
   a) (Number of outputs) T' must have n outputs. The same happens for T, since there is an $\mathsf{outlen}(\mathsf{rtx}) = n$ condition specified by the script in the same if statement that checks the branch.
   b) (Arguments) According to the compiler definition, the i-th output of T' will have arguments $\mathsf{nonce} = \mathbf{nonce_i}$, $\mathsf{name} = \mathtt{X}_i$, $\mathsf{branch} = j$, $\alpha_{\mathsf{l}} = a^i_l$, and $\beta_{\mathsf{l}} = b^i_l$. This coincides with the number of arguments of the i-th output of T, since the script scr_{D_j}, which T must satisfy, contains the term $\mathsf{arglen}(\mathsf{rtxo(i)}) = |\boldsymbol{\alpha^i}| + |\boldsymbol{\beta^i}| + 3$. The value of $\mathbf{nonce_i}$ has been chosen to be exactly equal to the first element of the i-th output of T, and this is also the first argument of T'. The same reasoning holds for the last $|\boldsymbol{\beta}^i|$, which were used in the construction of the values b^i_l. Regarding the remaining argument we can see that the script of $txout(x)$ forces each of them to have a precise value: the second (the branch argument) must be equal to j, the third (the name argument) must be equal to $\mathtt{X}_i$, and for all the other $m = |\boldsymbol{\alpha^i}|$ arguments we have the following constraint:

$$\mathsf{rtxo}(i).\alpha_1 = \mathsf{ctxo}.a^i_1 \ \mathsf{and}\ \cdots\ \mathsf{and}$$
$$\mathsf{rtxo}(i).\alpha_{\mathsf{m}} = \mathsf{ctxo}.a^i_m$$

   which appears in the last part of the script for a clause operation. Remember that the expression $\mathsf{ctxo}.a^i_l$ is a shorthand for whatever combination of parameters have been used to specify the value assigned to the variable α^i_l in C. However, we already know that these values must evaluate to a^i_l, since they are evaluated from the arguments of $txout(x)$ (which we know to be compiler generated and encoding C). This means that the arguments of each output of T are the same to the one of the corresponding output of T'.
   c) (Value) The i-th output of T' has value v_i. In the script for a branch that contains a call operation, we have the following term

$$\mathsf{rtxo}(i).\mathsf{val} = \mathsf{rtxo}(i).\mathcal{E}_i,$$

   where $\mathcal{E}_i$ is the expression in the precondition of $\mathtt{X}_j$. We know that $\mathcal{E}_i$ must evaluate to v_i, since we have $\mathtt{X}_i\langle \boldsymbol{a^i}; \boldsymbol{b^i}\rangle \equiv \{v_i\}\ C_i$. So, since T has to satisfy the script, the value of its i-th output must be v_i.
   d) (Script) The script of each output of T' is the same of its first input, which is $\mathbf{txout}(x)$. The script of $\mathbf{txout}(x)$ contains the covenant $\mathsf{verrec}(i)$ that forces the i-th output of any transaction who redeems it to be equal to its own. From this we can conclude that the script of each output of T coincides with the script of each output of T'
3) (Outputs - send) If $\bar{D}$ ends in a `send`, then we can use a reasoning similar to the `call` case to show that the outputs of T' must be equal to the outputs of T. Actually, the situation is even simpler, since in this case the redeeming transaction must only have two arguments. However we will not delve into the details to avoid excessively lengthening this already long proof.

□

Essentially, we have just shown that any transaction that can redeem an output representing an active contract can be represented symbolically with a continuation advertisement term. This result plays a fundamental role in the proof of the computational soundness theorem. We can take this correspondence between transactions and advertisements even further, by showing that if the computational transaction respects the timing conditions set in the coherence definition (in particular in item 3), then the corresponding symbolic advertisement is actually valid in the configuration.

Theorem 8. *Under the same hypotheses of Theorem 7, let* $\Phi = \left[\bar{D}\ ;\ \boldsymbol{z}\ ;\ w\ ;\ (x\,,j)\right]_h$ *be the continuation advertisement constructed in the proof. Then* Φ *is valid in* Γ_{R^s}.

Proof. We know that the deposits z_j are the pre-image under $txout$ of some outputs in $\mathbf{B}_{R^c}$. Moreover, these outputs are unspent so, by Proposition 4 they must actually appear in the configuration. Also, thanks to Propositon 6, and remembering how w was constructed, we know that w must be smaller or equal to the value stored in $\mathcal{D}$.

Then, we have the timing requirements: the time in the configuration must be so that all waiting decorations in $\bar{D}$ are satisfied. In the computational setting all of T timelocks are expired, and those same timelocks were subject to the script's constrictions, which in turn were based on the `after` and `afterRel` decorations of $\bar{D}$. Since the time increases in the same way in both models, the timing requirements are satisfied.

Then, we have a condition which states that the sum of the "output" values of $\bar{D}$ (meaning the funds of each clause if $\bar{D}$ ends in `call` and the values distributed to each participant if it ends in a `send`) must be greater than 0 and lower or equal to the sum of the inputs values (the deposits z_i, the value w and the balance of the contract x). Since these directly translate to inputs and outputs of T we do not need to prove anything.

The last condition for validity applies only if $\bar{D}$ ends in a `call` operation: the proposition p_i in each clause precondition must be satisfied. Again, the fact that T must satisfy a compiler generated script is enough to prove this condition, since by including

$$\mathsf{and}\ \mathsf{rtxo}(1).p\ \mathsf{and}\ \cdots\ \mathsf{and}\ \mathsf{rtxo}(n).p$$

the script ensures that all clauses are satisfied. □

G. Translating symbolic strategies

This appendix aims to construct an algorithmic map $\aleph$ that transforms an honest symbolic strategy Σ^s_{A} into a computational strategy $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$. By Definition 22 $\aleph(\Sigma^s_{\mathsf{A}})$ will be an algorithm that takes as input a computational run R^c and a randomness source r_{A}, and returns a set of computational labels Λ^c, while attaining to some constraints.

The general idea behind our construction of $\aleph(\Sigma^s_{\mathsf{A}})$ is the following: the algorithm will first parse R^c in order to create a symbolic run R^s, then it will use Σ^c_{A} to produce a set of symbolic actions, which will lastly be translated into computational labels, concluding the process. In this way, $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$ is emulating its symbolic counterpart Σ^s_{A}. These procedures closely resemble the definition of the coherence relation, so we will not present every detail.

Parsing the computational run. Here, we will take a consistent computational run R^c and parse it, in order to construct a symbolic run R^s coherent to it. This will be a step-by-step construction, that takes a single label and finds a corresponding symbolic action. While doing that, we update the maps $txout$ (between names and outputs), $prevTx$ (between advertisements and transactions), and key (between participants and their public key): these will helps us to keep track of the symbolic terms that we created.

We begin with the initial prefix of R^c, which contains a transaction T_0 followed by messages that transmit the computational participant's public keys. By looking at those messages, we create a set of symbolic participants, and the key that associates to each of them their public key. Then, by looking at the outputs of T_0 we obtain a series of deposits, which, together with an empty destroyed funds counter $\mathcal{D}(0)$, and the time $t = 0$, will form the initial symbolic configuration Γ_0, which will be the prefix of the symbolic run. The map $txout$ is constructed to map each deposit of Γ_0 to the corresponding output.

Then, we have different scenarios according to the next computational step λ^c. If λ^c is a message we ignore it, except for the following cases:

1) It is the encoding of a incomplete advertisement Θ, in which case we perform the symbolic step $msg(\Theta)$.
2) It is the encoding of a compiler generated transaction T_Φ, never sent before in the computational run (i.e. not belonging to $\operatorname{ran} prevTx$). In this case λ^c corresponds to the advertisement of the valid term Φ that has a subscript h never used in the symbolic configuration (and we update $\operatorname{ran} prevTx$).
3) It is the encoding of a transactions T that takes at least an input in $\operatorname{ran} txout$, is neither compiler-generated nor correspondent to a join divide or donate operation, and never sent before in the computational run. In this case λ^c corresponds to a destroy advertisement.
4) It is a quadruple (T, j, wit, i), where wit is the signature with A's key on the j-th output of T (a transaction that is already present as a message in the run). Moreover $prevTx^{-1}(\mathsf{T}) = \Phi$; and it is the first time λ^c is broadcast after a broadcast of T. In this case λ^c corresponds to a symbolic authorization step, either $auth - in(\mathsf{A}, z, \Phi)$ or $auth - act(\mathsf{A}, \Phi)$ depending on what kind of advertisement Φ is, and on which input of T is being signed.

If λ^c is a transaction with at least one input in $\operatorname{ran} txout$, then we can analyse its structure and find the corresponding action among the following: $init(\Phi)$, $call(\Phi)$, $send(\Phi)$, $join(x, y)$, $divide(x, v, v')$, $donate(\mathsf{B}, x)$, or $destroy(\Phi)$. Lastly if λ^c is a computational delay we directly translate it to a symbolic one.

Each step in this conversion process is uniquely determined, up to different choices for the names of participants, deposit, contracts, and h subscripts, meaning that the above paragraph can be seen as proof for this proposition:

Proposition 9. *Given R^c we can find R^s, $txout$, key, $prevTx$ such that*

$$coher(R^s, R^c, txout, key, prevTx).$$

Moreover if $\dot{R}^s$, $txout'$, key', $prevTx'$ are such that

$$coher(\dot{R}^s, R^c, txout', key', prevTx'),$$

then we can get $\dot{R}^s$ from R^s by substituting each name x with $txout'^{-1}(txout(x))$, each advertisement Φ with $prevTx'^{-1}(prevTx(\Phi))$, and each participant name A with $key'^{-1}(key(\mathsf{A}))$.

Randomness. Every strategy, symbolic or computational, takes as input a random seed r_{A}. In order provide a proper translation between strategies we need to make a few remarks on this randomness source. Every computational strategy needs to use its randomness source in order to produce the key pairs that will be broadcast at the start of the run. This action does not have any symbolic counterpart, since it is assumed that symbolic participants can give authorizations without needing to worry about the low level signature details. It is also very important that the random bits used for key generation are never reused when choosing which action to perform in later steps, since this would cause a correlation between the keys and the later outputs of the run, potentially leaking information about the secret keys. So, in order to avoid this problem, the symbolic strategy that we are trying to emulate must be prevented from seeing the part of the random sequence used in the keys generation process. For this reason we will split the sequence r_{A} in two, $\pi_1(r_{\mathsf{A}})$ and $\pi_2(r_{\mathsf{A}})$ where the first part can be used in strategies

and is given as input to the Σ^s_{A}, while the second is only used for the initial keys generation.

From symbolic actions to computational labels. Once we have converted R^c to R^s we can compute $\Lambda^s = \Sigma^s_{\mathsf{A}}(R^s, \pi_1(r_{\mathsf{A}}))$. Then, we can transform each element α of Λ^s into a computational label λ^c, by following the corresponding case inside the coherence definition. However we must ensure that the constraints posed in Definition 22 are respected. In the following paragraphs we will show how to do that. When calling the compiler $\ddot{\mathbb{B}}_{\mathbf{adv}}$ we will always assume that the auxiliary functions are the ones constructed by the parsing step.

Advertisements.

1) (Incomplete advertisement). If $\alpha = msg(\Theta)$, then λ^c is simply a message encoding Θ.
2) (Complete initial advertisement). Remember that by Definition 16 Σ^s_{A} can choose $\alpha = adv(\Phi)$ with Φ complete only if the advertisement has $w = \star$. If $\Phi = [\mathtt{X}\langle \boldsymbol{a}; \boldsymbol{b}\rangle \,;\, \boldsymbol{z} \,;\, \star]_h$ then we can compile it to T_Φ by choosing the compiler's inputs in the following way: $\mathbf{in_i} = txout(z_i)$ for all i; t_0 is the time in Γ_{R^s}; $\mathbf{t_1}$ is the biggest delay specified in a `afterRel` in D; $\mathbf{t_j} = 0$ for all $j > 1$; and $\mathbf{nonce}$ is a list of numbers chosen so that the compiled transaction T_Φ is different from any previously broadcast transaction. The corresponding computational label in this case is $\lambda^c = \mathsf{A} : \mathsf{T}_\Phi \to *$.
3) (Complete continuation advertisement). If $\Phi = \left[\bar{D} \,;\, \boldsymbol{z} \,;\, \star \,;\, (x, j)\right]_h$ then, similarly to the case above, we can construct T_Φ by setting the appropriate compiler inputs. We then have that $\alpha = adv(\Phi)$ corresponds to $\lambda^c = \mathsf{A} : \mathsf{T}_\Phi \to *$.
4) (Complete destroy advertisement). If $\alpha = adv(\Phi)$ with $\Phi = [\boldsymbol{z} \,;\, \star]_h$ then $\lambda^c = \mathsf{A} : \mathsf{T}_\Phi \to *$, where T_Φ is a transaction with inputs given by $txout(z_j)$ and an irredeemable output that has script $\mathsf{scr} = \mathsf{false}$.

Authorizations.

1) (Advertised actions). If $\alpha = auth - act(\mathsf{A}, \Phi)$ or $auth - in(\mathsf{A}, \Phi, z)$ then $\lambda^c = \mathsf{A} : m \to *$ where m is a quadruple $(\mathsf{T}_\Phi, j, wit, i)$ encoding the corresponding witness. Notice that since α can be sent only if Φ is in the configuration, there must have already been an action $adv(\Phi)$ in the symbolic run. The fact that R^s is obtained by parsing R^c, which is consistent, means that if this step is reached T_Φ is already present in the run.
2) (Deposits). If α is an authorization for a $join$, $divide$, or $donate$ action, then we are not sure if the corresponding transaction is already present in the run (since they do not require a symbolic advertisement). So, if there T is not in the blockchain then $\lambda^c = \mathsf{A} : \mathsf{T} \to *$. If instead it is already present we act like in item 1, with $\lambda^c = \mathsf{A} : m \to *$, and m encodes the required signature.

Actions.

1) (Advertised actions). If α is an $init$, $call$, $send$ or $destroy$ action, consuming a term Φ, then the corresponding computational label is $\lambda^c = \mathsf{T}_\Phi = prevTx(\Phi)$. Remembering again that in Φ we must have $w = \star$, we will prove that λ^c satisfies the constraints given to symbolic strategies. Indeed, for α to be possible the advertised term Φ and all the authorizations must be in the configuration Γ_{R^s}. Since R^s is constructed by parsing R^c this means that T_Φ has been sent on the computational run (after its timelocks are exhausted), and that all the witnesses that correspond to a symbolic authorization are present. However, since $w = \star$ these are all the needed witnesses and T may be choosen as an action by a computational strategy.
2) (Deposit actions).If α is a $join$, $divide$, or $donate$ action, then the corresponding computational label is $\lambda^c = \mathsf{T}$. Again, the parsing ensures us that T and all its witnesses are already sent in the computational run.

H. Security of the compiler

In this appendix we prove the main result of this paper: the security of the ILLUM compiler. We will see that if the participants choose their computational strategy by translating a symbolic strategy, then no matter what the computational adversary does, it is possible, with overwhelming probability, to simulate any of its computational action in the symbolic world, therefore maintaining coherence.

Theorem 10 (Security of the compiler)**.** *Let $\boldsymbol{\Sigma}^s$ be a set of computational strategies for all honest participants, and $\boldsymbol{\Sigma}^c$ be a set of computational strategies consisting of $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$ for all $\mathsf{A} \in \mathsf{Hon}$ and of an adversary strategy Σ^c_{Adv}. Given the security parameter η and any $k \in \mathbb{N}$, we define*

$$\begin{aligned} P(r) = \forall R^c \textit{ conforming to } (\boldsymbol{\Sigma}^c, r) \textit{ with } |R^c| \leq \eta^k, \\ \exists R^s, txout, key, prevTx, \textit{ such that} \\ coher(R^s, R^c, txout, key, prevTx) \textit{ holds} \\ \textit{and } R^s \textit{ conforms to } (\boldsymbol{\Sigma}^s, \pi_1(r)). \end{aligned}$$

then, the set $\{r | P(r)\}$ has overwhelming probability

Proof. Consider any given r, and take R^c that satisfies the conformance hypothesis and the length requirement. Assume also that there is no corresponding symbolic run R^s. We will show that this happens with negligible probability.

Take $\dot{R}^c$, the longest prefix of R^c such that there exist a corresponding run $\dot{R}^s$ and maps $txout$, key, and $prevTx$ for which $coher(\dot{R}^s, \dot{R}^c, txout, key, prevTx)$ holds. This $\dot{R}^c$ is not empty, since the initial prefix of R^c (consisting of T_0 and the broadcast of public keys) can always be transformed into a corresponding initial symbolic run (and the conformance with strategies trivially holds for initial runs). We will now proceed by cases on all the possible labels λ^c that can extend $\dot{R}^c$ to show that either it's possible extending $\dot{R}^s$ to a run coherent with $\dot{R}^c\lambda^c$ (reaching a contradiction), or that the adversary has managed to produce a signature forgery (which only happens with negligible probability).

1) $\lambda^c = \mathsf{B} \to * : m$. Looking at the coherence definition we can see that we need to consider four distinct cases for the message: (i) m encodes an incomplete

advertisement Φ; (ii) m encodes a transaction T_Φ, where Φ is a valid advertisement term; (iii) m is quadruple encoding a witness for an input (with a symbolic counterpart) of some T that was already broadcast in the run after its timelocks have expired; (iv) m is any other message. The first two cases are handled with a $msg(\Phi)$ and an $adv(\Phi)$ symbolic action respectively, and in the fourth case the symbolic run ignores the message m. This leaves us with case (iii) where the adversarial strategy has been able to produce a witness: this means either that it has forged a signature (and this happens with negligible probability), or that some honest A chose to provide it. However, if that's true, then the symbolic strategy of A must have enabled the authorization at some point, since $\Sigma^c_{\mathsf{A}} = \aleph(\Sigma^s_{\mathsf{A}})$. This means that Σ^s_{Adv} can choose it as next action α, meaning that $\dot{R}^s \xrightarrow{\alpha} \Gamma$ is still coherent with $R^c\lambda^c$.

2) $\lambda^c = \mathsf{T}$. Again, we have multiple cases.
   a) If T does not have any inputs in $\operatorname{ran} txout$ then coherence is achieved without adding any additional step to $\dot{R}^s$.
   b) If T has some inputs in $\operatorname{ran} txout$, and one of them is the image of an active contract, then, by Theorem 7 we have that T must be a compiler-generated transaction T_Φ. Since λ^c has been chosen by Σ^c_{Adv}, we know it must follow the rules for strategies, so T has been broadcast earlier (and not before its timelock are over), and all its witnesses have been broadcast too. This means that there has been a corresponding symbolic advertisement, (since $\dot{R}^s$ is coherent to $\dot{R}^c$), so Φ has been included in the configuration. Theorem 7 also tells us that Φ is in the form $\left[\bar{D}\,;\boldsymbol{z}\,;w\,;(x,j)\right]_h$ and Theorem 8 proves that Φ is valid in $\Gamma_{\dot{R}^s}$. Notice also that, thanks to the conditions on strategies, we know that T's witness have been broadcast in some previous step of the run, and, by coherence, this means that all the required symbolic authorizations are present in Γ_{R^s}. The validity of Φ and the presence of the deposit's authorization ensure that the continuation action corresponding to T (either $call(\Phi)$ or a $send(\Phi)$) can be performed in $\dot{R}^s$, meaning that we can extend $\dot{R}^s$ and still achieving coherence.
   c) If T has some inputs in $\operatorname{ran} txout$, but none of them is the image of any active contract, we have 3 possible situations: $\mathsf{T} = \mathsf{T}_\Phi$ is compiler-generated starting from an initial advertisement Φ; T is a transaction associated to a deposit action $join$, $divide$, or $donate$; or T is some other transaction. In the first case we can carry out the same reasoning of step (b) to conclude that $\alpha = init(\Phi)$ is a continuation that achieves coherence. In the second case we can achieve coherence by letting α be the corresponding symbolic deposit operation. In this case too all deposits and authorizations must be present in the run. In the third case we will choose α to be a destroy operation. Again, we notice that T must have been broadcast at some point, and since it is not a compiler generated transaction, nor it does correspond to a deposit action, the broadcast message falls into the case described by item 4 of the first inductive case of Definition 25; and the broadcast is thus mirrored in the symbolic run by the advertisement $adv(\Phi)$, where $\Phi = [\boldsymbol{z}\,;w]_h$. In this case too all of T witnesses must have been advertised, meaning that all authorization for deposits z_j are present in $\Gamma_{\dot{R}^s}$. This means that in this case too we can achieve coherence by extending $\dot{R}^s$ with $\alpha = destroy(\Phi)$.
3) $\lambda^c = \delta$. Here we can extend $\dot{R}^s$ with $delay(\delta)$. This trivially keep coherence between runs. Moreover the resulting symbolic run still conforms to the strategies: in the computational case all honest participant had to agree on the delay, which, by the definition of $\aleph$ implies that it is also the case for the symbolic strategies.

In each of these cases we manage, with overwhelming probability, to extend $\dot{R}^s$ to something that is coherent with $\dot{R}^c\lambda^c$ (against the maximality of the prefix $\dot{R}^c$), and this concludes our proof. □

```
contract Foo {
 int x;       // integer
 uint u;      // unsigned integer
 address a;   // address (externally-owned)
 mapping (address => uint) m;
 ...          // other state variables

 constructor(...) { ... } // Entry point

 function f(int x, ..., address b, ...)
  input(e:T) // Receive tokens
  auth(c)    // Authorizations
  after(t)   // Time constraint
  ...        // other modifiers...
 {
  int y;     // local variables
  ... // function body
 } next(g1,...,gn) // Possible continuations

 ...  // other functions

 function g(...) view { // pure function
  ... // return expression
 }
}
```

Figure 10: General form of HELLUM contracts.

I. Compiling HELLUM into ILLUM

We describe in this section how to compile high-level contracts written in HELLUM to the intermediate-level language ILLUM, as sketched in Section 7.

We start by providing more details about HELLUM, referring to https://github.com/bitbart/illum-lang/ for its concrete syntax and typing rules. A contract has a set of variables that define its state, and a set of functions with an imperative, loop-free body that can modify the contract state and transfer tokens. Base types comprise `bool`, `int`, `uint`, `string`, and `address`. Variables can also be `mapping`s from base types to base types. A function can have *modifiers* that must be satisfied before it can be called (as in Solidity), and *continuations* that specify which functions can be called after it. The general form of contracts is in Figure 10.

HELLUM functions have four possible modifiers:

- `after(t)` requires that the function is called only after (absolute) time `t`. Here, we abstract from the granularity of time: it could be e.g. a block number (as in Solidity) or a timestamp;
- `auth(a)` requires that the function call is authorized by address `a` (through `a`'s private key);
- `input(e:T)` requires that `e` tokens of type `T` are sent to the contract alongside with the function call, by any address;
- `next(g1 ... gn)` specifies the functions that can be called after the current function has been executed. When the `next` modifier is omitted, any continuation (except the constructor) is possible.

Note that the expressions appearing within the `after` modifier may only depend on the contract variables, while the expressions within `input` and `auth` may also depend on the function parameters. A function can use multiple instances of the same modifiers, except for `next`.

Function bodies are as in Solidity, but for the absence of loops and contract calls: they comprise assignments (to variables and mappings), sequences of commands, conditionals, and `require(e)` statements, which make the function fail when the expression `e` evaluates to false. The command `a.transfer(e:T)` transfers `e` units of token `T` to address `a`. Local variables, not contributing to the contract state, can be declared and used. Expressions are standard, and follow the Solidity syntax. The special expression `balance(T)` gives the number of units of token `T` currently available in the contract. Expressions can contain calls to pure functions (tagged as `view` in the contract). The HELLUM compiler includes a semantic analyzer that performs type checking and other checks to ensure the well-formedness of contracts.

Compilation: normal form. The first phase of the HELLUM compiler is a series of code transformations to bring contracts in the normal form described in Section 7. This phase is split into several steps:

1) macro-expand calls to pure functions into the corresponding expressions;
2) rewrite each function as a chain of conditional statements, and merge the `require` statements in a single `require` at the top of the function;
3) rewrite the body of each conditional branch in static-single-assignment (SSA) form [43], where each variable is written exactly once. Besides the contract variables, in this step we also add auxiliary variables keeping track of the varying contract token balances;
4) rewrite the body of each conditional branch so that the token transfers occur before all the assignments;
5) rewrite the body of each conditional branch so that all the assignments are folded into a single, simultaneous assignment of all the contract variables.

Below, we illustrate the code transformations 2 to 5 through a series of examples, referring to the repository https://github.com/bitbart/illum-lang/ for the full details and for the transformation from normal form contracts to ILLUM, as sketched in Section 7.

For step (2) of the normal form construction, we match patterns of the function body, and rewrite them to pull `require` statements out of conditional blocks, and push `transfer` and assignment commands within conditional blocks. These transformations modify the guards conditionals and other expressions preserving the semantics. We illustrate the patterns through code snippets, showing how the left part is transformed into the right part.

Payments and assignments before a conditional are pushed within the conditional, adapting the guards to match the state updates. For instance:

```
a.transfer(y:T);
if (balance(T)<7) {
  // c1
}
else {
  // c2
}
```

```
if (balance(T)-y<7) {
  a.transfer(y:T);
  // c1
}
else {
  a.transfer(y:T);
  // c2
}
```

```
x=y-5;
if (x<3) {
  // c1
}
else {
  // c2
}
```

```
if (y-5<3) {
  x=y-5;
  // c1
}
else {
  x=y-5;
  // c2
}
```

The commands (of any kind) after a conditional are pushed within all the conditional branches. For instance:

```
if (x<3) {
  // c1
}
else {
  // c2
}
x=x+1;
```

```
if (x<3) {
  // c1
  x=x+1;
}
else {
  // c2
  x=x+1;
}
```

Nested conditional statements are flattened:

```
if (x<=9) {
  if (x>5) {
    // c1
  }
  else {
    // c2
  }
}
else {
  // c3
}
```

```
if (x<=9 && x>5) {
  // c1;
}
else if (x<=9) {
  // c2
}
else {
  // c3
}
```

Each `require` command is moved to the top of the function by swapping it with the previous command, and updating the guard accordingly. For instance:

```
x=x+y;
require x<500;
```

```
require x+y<500;
x=x+y;
```

```
a.transfer(x:T);
require balance(T)>5;
```

```
require balance(T)-x>5;
a.transfer(x+y:T);
```

The most complex case is when a a `require` occurs in each branch of a conditional statement. In this case, we pull all the `require` out of the branches, and we combine the guards in the `require` commands with the guards of the conditional, obtaining a single `require`. For instance:

```
if (x<2) {
  require x>0;
  // c1
} else if (x<4) {
  require x>2;
  // c2
} else {
  require x<8;
  // c3
}
```

```
require ((x<2 && x>0) ||
  (x>=2 && (x<4 && x>2))) ||
  ((x>=2 && x>=4) && x<8);
if (x<2) {
  // c1
}
else if (x<4) {
  // c2
}
else {
  // c3
}
```

For step (3) of the normal form construction, we rewrite every conditional branch in SSA form. We do so by introducing an expression `balance_pre(T)` (which returns the amount of token `T` stored in the contract before the function invocation) as well as local variables when they are needed. For illustration, we consider a single branch and assume that `a`, `x`, `y` are global variables of the contract, while `z` is a function parameter.

```
y = x + balance(T);
x = z;
a.transfer(x:T);
a.transfer(y:T);
y = balance(T) + z;
```

The transformation introduces new local variables at each step, to keep track of the values of `a,x,y,z` and of the contract balance. For instance, the variables `x_i` are introduced at every assignment of `x`. A new variable `bal_T_i` is introduced upon each `transfer` to keep track of the balance of token `T`. Initially, we let `bal_T_i` to be `balance_pre(T)` (plus eventual function inputs). Our branch ends up rewritten as:

```
x_0,y_0,a_0,z_0,bal_T_0 =
x,  y,  a,  z,  balance_pre(T);
y_1 = x_0+bal_T_0;
x_1 = z_0;
a_0.transfer(x_1:T);
bal_T_1 = bal_T_0-x_1;
a_0.transfer(y_1:T);
bal_T_2 = bal_T_1-y_1;
y_2 = bal_T_2+z_0;
x,y,a,bal_T_fin = x_1,y_2,a_0,bal_T_2;
```

For step (4), we now move the two `transfer()` statements to the top by exchanging them with the assignments. To do this, we replace the variables appearing in `transfer()` with the expression on the right hand side of the assignment. In our example, we get:

```
a.transfer(z:T);
a.transfer(x+balance_pre(T):T);
x_0,y_0,a_0,z_0,bal_T_0 =
x,  y,  a,  z,  balance_pre(T);
y_1 = x_0+bal_T_0;
x_1 = z_0;
bal_T_1 = bal_T_0-x_1;
bal_T_2 = bal_T_1-y_1;
y_2 = bal_T_2+z_0;
x,y,a,bal_T_fin = x_1,y_2,a_0,bal_T_2;
```

For step (5), we collapse all the assignments into a single simultaneous one, that assigns the new values to the contract variables. In our example:

```
a.transfer(z:T);
a.transfer(x+balance_pre(T):T);
x,y,a,bal_T_fin = z,((balance_pre(T)-z)-(x+
    balance_pre(T)))+z,a,(balance_pre(T)-z)-(x+
    balance_pre(T));
```

From this last normal form, we can generate the ILLUM function clauses `f_run` and `f_next` as discussed in Section 7.

On loops in HELLUM. The HELLUM language does not feature loops. On the one hand, this makes the compilation to ILLUM easier, but on the other hand this reduces the expressivity of HELLUM. Allowing loops in HELLUM could be done in three ways. The simplest option is to extend the language with specific iterators on key-value maps (e.g., map, filter, fold). These operators could then be compiled in corresponding operators in a suitably extended ILLUM. More specifically, this would only require extending the ILLUM and UTXO script expressions with suitable operators. Since such loops would be bounded, this option does not strictly require a gas mechanism to prevent divergent behaviours. A second option would be to allow arbitrary (unbounded) loops in HELLUM and suitably extend the ILLUM expressions with operators that can simulate such arbitrary HELLUM loops (e.g., a fixed point operator). Note that such an extension would make the evaluation of ILLUM expressions potentially divergent, hence it would require a gas mechanism or some other means to bound the computation. For example, the Cardano scripting language (Plutus Core) is an untyped lambda calculus, thus allowing for unbounded computation, but the Cardano platform limits the execution of scripts to a given amount of computation steps. A last option would be to allow arbitrary HELLUM loops but compile them to a *chain* of recursive ILLUM clauses. Intuitively, calling such a recursive clause would only perform a part of the loop (say, the first iteration), and then call itself with the updated state. The recursion then stops whenever the loop is over, and proceeds to call another clause. While this mechanism effectively makes ILLUM Turing-complete, it requires the users to perform a potentially large number of calls, hence to append a large number of transactions on the blockchain, paying the fees for all of them. Further, this could lead to Denial of Service attacks. A malicious participant could call a HELLUM function which performs a long loop, pay the fees for the first few iterations and then stop interacting. In this way, the other participants are prevented to call other methods until they first complete the long loop by paying all the fees themselves. Worse, there is nothing stopping a malicious participant from calling the method again after its completion, blocking honest users from accessing the contract and forcing them to pay the fees once again. Therefore, this last option for handling loops would require more complex protocols to counter attacks like the ones described above.