Formal Aspects of Smart Contract Security

Thesis submitted in fulfilment of the requirements of the Doctoral Program in Mathematics at the University of Trento

PhD Candidate: Riccardo Marchesin
Supervisor: Prof. Roberto Zunino
Co-supervisor: Prof. Massimo Bartoletti

July 20, 2026

Contents

Chapter 1

Introduction

In this thesis I will present the research conducted during my doctoral studies. As the title implies, the overarching topic of this thesis is the security of smart contracts, which are agreements stipulated through an automated program. The context in which we we will consider those agreements is that of blockchains, decentralized ledgers that record transactions.

More specifically, the topic of this thesis is developed in two main lines of research: one regarding smart contract languages for UTXO blockchains and another one on the formalization of MEV attacks.

Some of the contributions presented in this thesis have been published in these conference papers and articles: [27, 30, 80], while one of has been submitted and is currently awaiting reviews [28].

In this section we will give a short overview of each chapter.

1.1 Contracts languages for UTXO blockchains

In this part of the thesis, we present concepts related to developing contracts in the UTXO model.

In general, writing contracts for UTXO-based blockchain is a challenging task, since the main object used to store currency, the *transaction outputs* do not lend themselves to a simple representation as objects in a classic procedural programming style. The chapters in this part deal with the design of contract languages for different UTXO models.

- The first chapter builds a simple intermediate-level language for smart contracts that is compiled using covenants, a primitive available to

some UTXO blockchains.

- The second chapter is linked to the first, and builds a high-level language that compiles down to the intermediate-level one mentioned above. This makes it easier to express meaningful contracts without having to handle the structures that are present at the lower level.
- The third chapter addresses the problem of scalability of UTXO contracts, and it proposes a more complex blockchain model and how to create contracts that are amenable to parallelization.
- Finally, the fourth chapter proposes a protocol for building simple off-chain contracts over Bitcoin, a UTXO blockchain whose expressive power is quite limited.

In the following paragraphs we will give a more detailed presentation of each chapter:

Intermediate language for contracts in the UTXO model. This section opens the chapter on UTXO-style contracts. It introduces Illum, an Intermediate-Level Language for the UTXO Model. Currently, blockchains following the UTXO model either provide contracts with limited expressiveness (Bitcoin), or require complex run-time environments (Cardano). Illum is a language that can express real-world smart contracts (e.g. those found in Decentralize Finance) but can still be compiled on minimal blockchains. We define a compiler from Illum to basic UTXO blockchains that feature *covenants*, a primitive that enables general state transitions in UTXO smart contracts. 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.

Acknowledgement of contribution: In this section I contributed to the theoretical side: both to the development of the language syntax and semantics, and to the proofs of security.

High level language for contracts in the UTXO model This section expands on the previous one. While the intermediate level language provides a much more convenient representation of smart contracts than the lower-level blockchain scripts, Illum is still far from being a human-friendly smart

contract language. To address this issue, we designed a companion language HeLLUM, which is a high level language inspired by Solidity that can be compiled down to Illum. We also show how a number of real-world smart contracts are implemented in HeLLUM.

Acknowledgement of contribution: In this section I contributed to the development of the HeLLUM-to-Illum compiler, as well as the design of the language. I also had a part in producing some examples.

Scalable UTXO contracts This section explores contracts in an "hybrid" UTXO model, that retains the main features of UTXOs while adopting an account system to keep track of the contract funds. Our motivation for choosing this approach lies in the fact that UTXO-based smart contract platforms face an efficiency bottleneck, in that any transaction sent to a contract must include the entire updated contract state. This requirement becomes particularly burdensome when the contract state contains dynamic data structures, which are needed in many use cases to track interactions between users and the contract. The problem is twofold: on the one hand, storing a large state in transactions implies a large transaction fee; on the other hand, a large centralized state is detrimental to the parallel validation of transactions - a feature that is often cited as a key advantage of UTXO-based blockchains over account-based ones. In the section, we propose a technique to efficiently execute smart contracts on an extended UTXO blockchain, which allows the contract state to be distributed across multiple UTXOs. We also propose a simple contract language (hURF) that compiles to the hybrid UTXO blockchain. In our approach, transactions only need to specify the part of the state they need to access, significantly reducing their size (and fees). We show how to exploit our model to parallelize the validation of transactions on multi-core CPUs. We implement our technique and provide an empirical validation of its effectiveness. Our experiments show that our parallel validation algorithm can frequently execute multiple calls in paralel, even when they affect the same smart contract.

Acknowledgement of contribution: In this article I contributed to the design of the hybrid UTXO platform and of the contract language hURF. The sections on implementation and experimental evaluation are included for completeness in the thesis, but they are mainly the work of my co-authors. In those sections I still contributed to the writing and presentation of the results.

Offchain contracts in Bitcoin In this section we consider the execution of smart contracts over Bitcoin "as-is", without assuming any extension such as covenants. This choice limits the type of contracts that can be expressed, but some simple examples (e.g. hashed time-locked commitments) can still be implemented. In this kind of contracts, every smart contract execution step corresponds to appending to the blockchain a new transaction that spends the output representing the old contract state, creating a new one for the updated state. This standard on-chain execution mechanism requires the contract participants to pay transaction fees for every execution step. In this section, we introduce a protocol that moves most of the execution of a Bitcoin contract off-chain. When all participants follow this protocol, they are able to save on transaction fees, drastically reducing them. By contrast, whenever adversaries try to disrupt the off-chain execution, any honest participant is still able to enforce the correct contract behaviour, by continuing its execution on-chain.

Acknowledgement of contribution: In this section I contributed to formalizing the protocol, starting from the work that contributed to the Master's thesis of my co-author D. Maddaloni. My work allowed to present the contributed by reframing the theory and presenting it in a short and self contained version.

1.2 Formalization of MEV attacks

The second part of the thesis focuses on Maximal Extractable Value (MEV) attacks, a class of attacks that can be performed by blockchain validators when selecting the proposed transaction to form a block. This problem is caused by the fact that different arrangements of transaction in a block may lead to a smart contract exchanging a different amount of tokens, which creates the incentive for a validator to *extract* as much value as possible from a block by including their own transactions. The primary target of these attacks are Decentralized Finance (DeFi) protocols, as their logic depends critically on transaction sequencing. To date, MEV attacks have already extracted billions of dollars in value, underscoring their systemic impact on blockchain security. This part includes a single chapter:

MEV formalization in Lean Verifying the absence of MEV attacks requires determining suitable upper bounds, i.e. proving that no adversarial

strategy can extract more value (if any) than expected by protocol designers. This problem is notoriously difficult: the space of adversarial strategies is extremely vast, making empirical studies and pen-and-paper reasoning insufficiently rigorous. In this section, we present the first mechanized formalization of MEV in the Lean theorem prover. We introduce a methodology to construct machine-checked proofs of MEV bounds, providing correctness guarantees beyond what is possible with state-of-the-art techniques. To demonstrate the generality of our approach, we model and analyse the MEV of the Automated Market Maker (AMM), a paradigmatic DeFi protocol. Notably, this is the first machine-checked proof of the optimality of the arbitrage and sandwich attacks in AMMs. Completing this mechanized proof was a challenging task, and required 7000 lines of Lean code.

Acknowledgement of contribution: In this section I contributed to the Lean formalization of definitions and proofs.

Statement on the use of AI

No AI tools were used in the development of the contributions presented in this thesis or in the writing.

Part I

Background

Chapter 2

Smart Contracts

Introduced in [109], smart contracts are, in Nicholas Szabo's words, "a set of promises, specified in digital form, including protocols within which the parties perform on these promises". Despite the generality of this definition it is easy to point the most prominent application for the notion of smart contracts: the implementation of financial services in cryptocurrencies platforms. This is due to the fact that blockchain technology, which is the underlying architecture of these platforms, facilitates the creation of programs that allow users to create programs that mediate the exchange of their valuables, permitting the specification and execution of logical instructions in a trustless environment.

2.1 Blockchains

Blockchains are data structures that maintain an ordered set of transactions, grouped together into blocks. The order relation between blocks, which creates the *chain* that gives name to the data structure, is formed by including an hash of the contents of the most recent block into the structure of a newly proposed one. This inclusion makes the blockchain hard to tamper with, since modifying the data included in a block would also change the block's hash, which in turn would cascade into changing all the subsequent blocks. Therefore, it is simple to check if there has been an attempt to "change the past" of the chain: one can do so by simply computing the blocks' hashes.

Blockchains are maintained by a network of participants (called nodes). Each node keeps a local copy of the blockchain. Nodes follow a *consensus al-*

gorithm that enables them to decide together which new block of transactions to include in the blockchain.

Consensus

When a new block of transactions is proposed for inclusion in the blockchain, the nodes must agree on it, not only to verify that it is valid (e.g. each transaction is allowed to spend its input funds, there is no double spending, smart contract conditions are respected), but also to agree on the order that the blocks are appended to the chain. This process is done through a consensus algorithm. In the history of blockchains, many consensus protocols have been proposed. For the sake of example, we mention the two of main classes of consensus protocols:

Proof of Work The first mechanism for determining consensus was *Proof of Work* (used by Bitcoin and early Ethereum versions). Here the nodes that participate in the consensus algorithm are known as *miners*. The power to determine the next block is awarded to the miner that first solves an hard cryptographic puzzle (e.g. finding a seed that, when hashed, yields a small value). In this way, the more computational power a miner accrues, the higher the chance that they can decide on the next block. To incentivise nodes to participate in the mining process, every block also pays a fee to the miner who appended it to the blockchain.

This safety of this consensus protocol hinges on the assumption that no miner will be able to get more than 50% of the computational resources of the whole network. Should this scenario occur, that miner would be able to modify the blockchain freely, appending invalid transactions faster than the other miners can correct them. This is known as the the 50% +1 attack.

Proof of Stake Another widely used consensus protocol is the so called *Proof of Stake* (used by Cardano, Algorand, and newer versions of Ethereum). One of the reasons for its introductions have been lowering the energy cost associated with being a miner in a Proof of Work blockchain. The nodes that participate to consensus here are called *validators*. Each validator choses to lock some amount of their currency, using it as a stake to acquire voting power for the validation of a newly proposed block. One of the validator is then chosen at random to propose the next block. Other validators then vote

to determine if the proposed block should be rejected (i.e. if it contained any illegal transaction). If an illegal block has been proposed, the protocol may punish who validated it by taking away its stake.

2.2 Smart Contract Models

Two main smart contracts models have emerged so far.

In the *account-based model*, the state of the blockchain is held in *accounts* that store both the amount of currency available in users' wallets and contracts' balances and the contract data. Transactions do not contain the entire state of a contract, but only the calls to contract's methods. Users are then able to reconstruct the current contract state by following the history of called methods. Contracts in this model are reactive objects that live on the blockchain: they have a state that can be accessed and modified by methods, as in object-oriented programming. These methods can be invoked by transactions that can modify the contract's state and transfer crypto-assets according to the programmed logic. The account-based model was introduced in 2015 by Ethereum, where smart contracts were popularized. Most blockchain platforms today follow the account-based model: besides Ethereum, also other main-stream blockchains such as Solana, Avalanche, Hedera, Algorand and Tezos are account-based.

In the *Unspent Transaction Output model* (UTXO), instead, the state of the blockchain is entirely maintained within transactions. Each transaction stores currency and data in its *outputs*, and each output sets the conditions that have to be met for it to be redeemed as *input* of a future transaction. Whenever a new transaction is published in the blockchain, it replaces ("spends") the output of an old transaction, effectively updating the contract state and the assets ownership. Transaction outputs associated to a contract must specify the assets they control, the contract state, and the condition under which those assets can be transferred again. For this reason programming contracts in the UTXO model requires a paradigm shift from the common object-oriented style [41]. The UTXO model was first proposed by Bitcoin. Despite being less widespread than the account model, other blockchains, such as Cardano and ZCash, have also adopted it. Since UTXO blockchains will be the focus of a main part of this thesis, a more detailed explanation of their structure can be found in Chapter 3.

Account-based vs. UTXO blockchains To give a quick idea of the two accounting models, we will use as an example a simple banking contract.

In the context of an account based blockchain, a user A who wants to withdraw `Bank` contract, will send a transaction to it, requesting to perform the `withdraw` method, and providing the signature needed for authentication. Once the transaction is appended to the blockchain, the method will be executed, updating both the state of the contract and the user's wallet.

Instead, in a UTXO-based blockchain, the state of the `Bank` contract could be scattered among a set of outputs. To withdraw currency from the bank A must specify which contract outputs they are going to spend, and produce a transaction that uses them as inputs, signing it to authenticate themselves. Among the outputs produced by this transactions, there will be one whose redeeming condition only allows it to be spend by A (e.g. by requesting a signature verification against A's public key) and one that contains the new state of the `Bank` contract. The redeeming conditions of this output associated to the `Bank` will then allow users to perform further Bank operations.

2.3 Threat model

In the next chapters we will consider different topics related to the security of smart contracts. Here we'll mention a few security assumptions that are common to the sections. We see the adversary as a set of malicious nodes who act as miner or validators. The adversary is therefore able to craft transactions with the knowledge of what transaction have been proposed and are currently in the mempool. The adversary is also able to perform scheduling attacks, delaying a user transaction or inserting their own transactions in between the ones that are proposed by users. Despite this, we assume a liveness property: if a users sends a valid transaction to the mempool, then it will be included in a block within a reasonable time frame. We justify this property by assuming that the malicious nodes do not constitute the totality of miners, meaning that even if the adversary ignores a user's transaction, the other nodes will end up including it in a block.

Finally, we assume that the underlying blockchain infrastructure over which contracts are executed is secure: we will not consider attacks involving the network level or the consensus level.

Chapter 3

Contracts in the UTXO Model

In this section we present the UTXO model of blockchain, which will be the basis for the first part of the thesis.

3.1 Basic UTXO: Bitcoin's model

The Unspent Transaction Output model, or UTXO model, was adopted by the first blockchain, Bitcoin. In this model, currency is spent by producing *transactions*, made up of multiple *outputs* (or UTXOs), each specifying a certain quantity of coins and a condition that has to be met in order to redeem them and spend them in a future transaction. These conditions are expressed with as a program, the *script*. A transaction that wants to spend the value stored in an output (*redeeming* it) has to specify it as one of its *inputs*, and it needs to provide a sequence of *witnesses* that are passed as arguments to the output's script, which is then evaluated. For a transaction to be valid, these evaluations have to yield true for all of the transaction's inputs.

In the original Bitcoin blockchain, the language for specifying scripts is rather limited: it can perform basic logical and mathematical operations, check whether a certain amount of time has passed, and verify signatures on the hash of the redeeming transaction (proving that it has been authorized by the private key owner). In the original Bitcoin protocol, the scripting language does not feature loops, nor it allows to access information about the redeeming transaction other than the signature field. This scripting language is not Turing complete, but it still allows for the design of meaningful smart

contracts, such as hashed time locked commitments and payment channels.

Transactions and blocks also include some additional fields: we are interested in the ones that relate to timing, since, as we will see, the script may also check that they respect certain temporal conditions.

Definition 1 (Transaction)**.** A transaction T is defined as a 5-uple (in, wit, out, $\mathsf{absLock}$, $\mathsf{relLock}$) 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 $(\mathsf{value}, \mathsf{script}, \mathsf{arg})$, where arg is a list of integers, script is a script (its syntax will be specified in the next paragraphs), and value is an integer.
- $\mathsf{absLock}$ is the absolute timelock, and it is a non negative integer.
- $\mathsf{relLock}$ is the list of relative timelocks. It has the same lenth as in and each of its elements is a non negative integer.

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

Going more into details about the field of transactions, we first consider the output field t. Each output in the list has a value field representing its value in bitcoins; a script field, which is the script expressing the conditions that need to be met in order for the output to be spent; and an arg field, which is a sequence of arguments (integers or bitstrings) used to encode a state in each output. The input list is used to denote the transaction outputs from which T takes the funds. In particular, if (T', k) appears in $\mathsf{T}.\mathsf{in}$, this means that T is spending the k-th output of T'. The witnesses and relative timelocks are related to the input list. In particular, if $\mathsf{T}.\mathsf{in}[j] = (\mathsf{T}', k)$, then we have that $\mathsf{wit}[j]$ is the sequence of arguments passed to $\mathsf{T}'.\mathsf{out}[k].\mathsf{script}$ in order to make it evaluate to true; while $\mathsf{relLock}[j]$ represents the amount of time that needs to pass (after the inclusion of T' in the blockchain) before T can be included. The absolute timelock $\mathsf{absLock}$ specifies a global time before which the transaction T can not take place.

We will now define the syntax and semantics of regular Bitcoin scripts.

Definition 2 (Scripts syntax)**.** The script field of a transaction output has the following syntax:

$e ::= v$	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 transaction input
$\mid \lvert e \rvert$	size (in bytes)
$\mid H(e)$	hash
$\mid$ 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

We also use some shorthands for common logical operations, setting (i) $true \stackrel{\text{def}}{=} 1$, (ii) $false \stackrel{\text{def}}{=} 0$, (iii) e and $e' \stackrel{\text{def}}{=}$ if e then e' else $false$, (iv) not $e \stackrel{\text{def}}{=}$ if e then $false$ else $true$, (v) e or $e' \stackrel{\text{def}}{=}$ 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 assuming that T.in$[i]$ is the redeeming input, unless specified otherwise. The full semantics for the evaluation operator can be found in [13]. First, we denote with $\bot$ a failure in the evaluation. As an example, the evaluation of $e.n$ when e is a sequence with less than n terms fails and yields $\bot$. All the operators in the script's syntax are *strict*, meaning that their evaluation yields $\bot$ if one of their arguments is $\bot$. The terms that we will highlight are rtxw, versig, absAfter, and afterRel.

- $[\![$rtxw$]\!]$ evaluates to T.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 $\bot$) against the key $[\![e]\!]$, and to $false$ otherwise.

- $[\![\mathsf{absAfter}\ e\colon e']\!]$ evaluates to $[\![e']\!]$ if the field absLock of the redeeming transaction T is greater or equal to $[\![e]\!]$, otherwise it evaluates to $\bot$. Similary, $[\![\mathsf{relAfter}\ e\colon e']\!]$ evaluates to $[\![e']\!]$ if the field relLock$[i]$ of T is greater or equal to $[\![e]\!]$, otherwise it fails, yielding $\bot$.

Example of a transaction Here we see an example of a simple transaction

T_1
in[1]: $\{\mathsf{T}, i\}$
wit[1]: $:w$ // signature for the first input
$\cdots$
out[1]: $\{$arg: $\cdots$,
script: versig(pk_{A}, rtxw), // verify signature
value: $1\}$
absLock: 3

In order to redeem the token held in the single output of this transaction, T.out[1], a transaction T″ has to satisfy the spending condition T.out[j].script, which simply amounts to signing T″ with the public key of A. This proves that A has authorized the spending of this output.

3.2 UTXO with Covenants

A natural extension for the Bitcoin UTXO is to allow *covenants* [24], a set of primitives that can be used by a script to access the outputs of the redeeming transaction, and to force their scripts to have a given value. This interaction between the scripts and the redeeming transaction's outputs allows us to preserve scripts across a chain of transactions, allowing recursion in contracts.

Definition 3 (Covenants)**.** We extend the script field of a transaction witht the following fields:

$e ::= \cdots$	previous expressions
$\mid \mathsf{verscr}(e, n)$	simple covenant
$\mid \mathsf{verrec}(e)$	recursive covenant

Covenant operators allow the script in T to constrain the scripts in T″. The simple covenant verscr(scr, n) mandates the n-th output of T″ to have a

script equal to scr. The recursive covenant 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.

Since covenants requires us to refer to scripts other than the one being executed, we introduce a few shorthands for our notation: First, ctxo refers to the current transaction output, i.e. the one that is being spent. We refer to a field f of that output with ctxo.f. We also have a way to refer to outputs of the *redeeming* transaction: like above, rtxo(e).f denotes the field f of the e-th output of the redeeming transaction.

To improve readability, we will use names instead of indices when referring to arguments in the arg sequence (e.g., we write ctxo.owner for ctxo.arg.1).

As an example of a transaction that makes use of covenants, consider the following T_2, which redeems the only output of T_1:

T_2

```
in[1]:  (T1, 1)
wit[1]: sig_A(T2)
out[1]: {arg: pk_A,
         script: versig(ctxo.arg.1, rtxw) and   // verify signature
                 rtxo(1).value = 1T and         // preserve value
                 verrec(1),                     // preserve script
         value: 1T}
```

The transaction T_2 can redeem its input, since it carries a witness (a signature of A) that satisfies the script $\mathsf{T}_1.\mathsf{out}[1].\mathsf{script}$. 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 script. 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 1T value (here 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 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.

Part II

Contributions: Smart Contract languages for UTXO blockchains

Chapter 4

An Intermediate-Level Language for the UTXO Model

4.1 The Illum intermediate 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:

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

Here, $\mathtt{X}$ is the clause name, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ 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 a few examples.

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 `if` *true* in the funding precondition.

To start the game, participants can invoke the `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 `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:

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

For instance, assume that $\texttt{Play}_{\mathsf{A}}\langle v;w\rangle$ is active when a player executes $\texttt{call}\ \texttt{Play}_{\mathsf{B}}\langle w;?\rangle$. 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 $\texttt{Play}_{\mathsf{B}}$, 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:

- $\texttt{send}\ (v_1\mathrm{T}_1 \to \mathsf{A}_1 \| \cdots \| v_n\mathrm{T}_n \to \mathsf{A}_n)$. 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.

- $\texttt{call}\ (\mathtt{X}_1\langle \boldsymbol{\mathcal{E}_1};\mathbf{?}\rangle \| \cdots \| \mathtt{X}_n\langle \boldsymbol{\mathcal{E}_n};\mathbf{?}\rangle)$. 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 $\mathbf{?}$ represent values chosen by participants. The `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 ($\texttt{after}\,t : D$), making the branch D enabled since a certain time t, or relative ($\texttt{afterRel}\,\delta : D$), 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., A : 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 $\boldsymbol{v}\mathrm{T}$. Note that $\boldsymbol{v}\mathrm{T}$ can be a sequence $v_1\mathrm{T}_1 \cdots v_n\mathrm{T}_n$, 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:

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

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 $6v\mathrm{T}$ overall. Since the contract balance is $v\mathrm{T}$, participants must provide $5v\mathrm{T}$. In practice, the owner A will need to convince two participants B and C to provide $2.5v\mathrm{T}$ 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 $2v\mathrm{T}$. If B and C later manage to enrol two other participants in the scheme, they will receive $4v\mathrm{T}$, gaining $4v\mathrm{T} - 2.5v\mathrm{T} = 1.5v\mathrm{T}$. 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.

Turing-completeness ILLUM is Turing-complete: indeed, we can simulate in ILLUM any counter machine [58], a well-known Turing-complete computational model. The proof is similar to that in [25]: 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.

4.2 Symbolic model of Illum contracts

In this section 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 4.1.

In this section we will formalize more precisely the syntax of Illum, expanding on the concepts presented in the previous pages.

Notation 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 α_i and β_i, while the actual values substituted to those parameters will be denoted by a_i and b_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 $\mathcal{N}_i$.

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 role is crucial to Illum constructs. Note that, in principle, it is easy to extend Illum by allowing parameters to also take the type of key-value mappings. The inclusion is not of interest from a theoretical point of view, so, to slightly simplify the presentation, in this chapter we will not include mappings among the possible parameters. On the other hand, adding mappings enables us to discuss more complex contracts: for this reason, in the next chapter, which is focused on creating a high-level language to abstract Illum (see Section 5.1), we will assume mappings to be available as a type of parameters.

Simplifications To improve readability, in the technical treatment we slightly simplify the model presented in the overview. 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. As mentioned above, we also omit mappings, since they are not strictly needed in the definition of the compiler, and could be easily added to the model.

4.2.1 Syntax of Illum contracts

Definition 4 (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 *process*. The clause takes two sequences of parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Parameters are of two types: integers and participants. We require that all the parameter names in $\boldsymbol{\mathcal{E}}$, p, and C are present in $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$.

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 $\{\boldsymbol{v}\ \mathtt{if}\ p\}$ encodes the requirements for the invocation of X. Namely, the number v asks the participants to transfer v tokens to the process C. Moreover, p is a boolean condition on the parameters that must hold. We write $\{\boldsymbol{v}\}$ for $\{\boldsymbol{v}\ \mathtt{if}\ \mathit{true}\}$.

Definition 5 (Processes)**.** Processes have the following syntax:

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

where we assume that: (i) each clause name X has a unique defining equation $\mathtt{X}(\boldsymbol{\alpha}\,;\boldsymbol{\beta}) = \{\boldsymbol{\mathcal{E}}\mathrm{T}\ \mathtt{if}\ p\}\ C$; (ii) the sequence $\boldsymbol{a}$ of actual parameters passed to

a called clause $\mathtt{X}_i$ matches, in length and typing, the sequence $\boldsymbol{\alpha}$ of formal (internal) parameters; (iii) the order of decorations is immaterial, (e.g. $\mathsf{A} : \mathtt{after}\, t : D$ is identified with $\mathtt{after}\, t : \mathsf{A} : D$).

Instantiated clauses, evaluation and closed form A clause $\mathtt{X}(\cdots)$ together with two correctly typed sequences of actual parameters $\boldsymbol{a}$ and $\boldsymbol{b}$ is said to be an *instantiated clause*, and denoted by $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle$.

Given a clause $\mathtt{X}(\boldsymbol{\alpha}\,;\boldsymbol{\beta}) = \{\mathcal{E}\ \mathtt{if}\ p\}\ C$ we write $\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle \equiv \{v\}\ C'$ if the following conditions hold:

(i) $[\![\mathcal{E}\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = v$;

(ii) $[\![p\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = \mathit{true}$;

(iii) $[\![C\{\boldsymbol{a}/\boldsymbol{\alpha}, \boldsymbol{b}/\boldsymbol{\beta}\}]\!] = C'$.

where $[\![\cdot]\!]$ is a simple arithmetic and logic evaluation operator. After this evaluation, every expression inside C' is reduced to an constant. Such a process is said to be in *closed form*. Unless specified otherwise, from this point onward we will be working with processes in closed form.

4.2.2 Configurations and semantics of Illum contracts

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

Besides these terms, in a configuration we also have advertisements and authorizations.

Advertisements terms are used by a participant to propose the activation of a new contract, the continuation of an active contract, or the destruction of a set of deposits. Advertisements may also be incomplete, in which case they are simply used to communicate messages between participants.

Authorizations 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.

In the next paragraphs we will see in detail how authorizations and advertisement terms are constructed.

Definition 6 (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:

$$
\begin{array}{rll}
\tilde{\Gamma} ::= & \langle C, v \rangle_x^t & \text{active contract} \\
\mid & \langle \mathsf{A}, v \rangle_x & \text{deposit} \\
\mid & \Phi & \text{advertisement} \\
\mid & \Theta & \text{incomplete advertisement} \\
\mid & \mathsf{A}[\chi] & \text{authorization} \\
\mid & (\tilde{\Gamma} \mid \tilde{\Gamma}) & \text{parallel composition}
\end{array}
$$

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).

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.

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 4.2 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 by their owner. We keep track of the currency that has been destroyed with a term in the configuration, the destroyed fund counter $\mathcal{D}(w)$. We assume that the funds in $\mathcal{D}(w)$ are only available to dishonest participants, and that they can spend them freely, without the need to produce any authorization term.

Advertisements in general Advertisements are terms included in the configuration by a participant who wants to propose an action (i.e. activate a new contract, make a contract step, or destroy deposits) to the others. These terms can be *incomplete* (denoted with Θ), and only used as a message between participants, or *complete* (denoted with Φ) which are used by the ILLUM contract semantics. in order to advance contract execution. In the following paragraphs we will describe precisely how each type of advertisement term is structured. After that, we will mention what are the parts that can be left unspecified to obtain an incomplete advertisement.

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 [1]. The subscript $h \in \mathbb{N}$ is just a nonce, used to differentiate two otherwise identical terms. In an initial advertisement the clause `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.

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 $\Phi = \left[\bar{D} \ ; \ \boldsymbol{z} \ ; \ w \ ; \ (x, j)\right]_h$. Like in the previous case, $\boldsymbol{z}$ and w are used to specify the source of additional funds used in the continuation action (note that $\boldsymbol{z}$ might 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

[1] 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.

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:

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

with $a^i_j \in \mathbb{Z} \cup \mathsf{Part}$ and $b^i_j \in \mathbb{Z} \cup \mathsf{Part} \cup \{*\}$.

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.

Incomplete advertisements (Messages) 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 $\Theta = \big[\bar{D}\ ;\ \boldsymbol{z}\ ;\ w\ ;\ (x\,,j)\big]$. 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 $\chi = \cdots \triangleright \cdots$ 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. $z \triangleright \Phi$, where z is a deposit, is used to authorize the use of z to fund the action advertised by Φ.

2. $x \triangleright \Phi$, 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. $z_1, z_2, i \triangleright v_1 + v_2$ 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. $z \triangleright v, v'$ is used to authorize the splitting of deposit $\langle \mathsf{A}, v + v'\rangle_z$ into two, of value v and v' respectively.

5. $z \triangleright \mathsf{B}$ is used to authorize the transfer of deposit $\langle \mathsf{A}, v\rangle_z$ to a participant B.

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.

Introducing Illum semantics Now that we have a full picture of the ILLUM syntax we can look more in detail at how contracts are executed. In the following paragraphs we will show the execution of a simple contract: this showcases the semantics of ILLUM, and the role of advertisements and authorizations terms.

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

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:

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

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

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

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 $\Phi_1 = [\mathsf{B} : \mathtt{call}\; \mathtt{X}\langle 1\,;3\rangle \;;\; z_2 \;;\; (x\,,2)]_h$. 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:

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

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:

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

and then withdraw the contract balance:

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

Definition 7 (Semantics)**.** The operational semantics of ILLUM is a labelled transition system between configurations, defined by the rules in Figure 4.1 and Figure 4.2.

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

$$\frac{\begin{array}{c}\Gamma \text{ contains } \Phi \text{ but not } \mathsf{B}[z \rhd \Phi] \\ \Phi \text{ is valid in } \Gamma \text{ and funded with deposit } z\end{array}}{\Gamma \xrightarrow{auth-in(\mathsf{B},z,\Phi)} \Gamma \mid \mathsf{B}[z \rhd \Phi]} \qquad \frac{\begin{array}{c}\Gamma \text{ contains } \Phi \text{ but not } \mathsf{A}[x \rhd \Phi] \\ \Phi = \left[\bar{D} \,;\, \boldsymbol{z} \,;\, w \,;\, (x, j)\right]_h \text{ is valid in } \Gamma \quad \bar{D} \approx \mathsf{A} : D'\end{array}}{\Gamma \xrightarrow{auth-act(\mathsf{A},\Phi)} \Gamma \mid \mathsf{A}[x \rhd \Phi]}$$

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

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

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

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

Figure 4.1: Semantics of ILLUM contracts.

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

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

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

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

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

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

Figure 4.2: Semantics of ILLUM deposits.

4.3 The Illum compiler

In this section we give the full definition of the compiler, which 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 $\mathbb{B}_{\mathbf{adv}}$.

We remind the reader that ILLUM is compiled down to a covenant-enabled UTXO blockchain, as presented in Section 3.2.

We begin by defining deposit outputs: these are transaction outputs with a fixed structure. They will be used to represent symbolic deposits in the computational model.

Definition 8 (Deposit output)**.** An output o is said to be a *deposit output owned by* A when it has exactly three arguments, the following script

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

and the third argument is equal to A.

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 `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 `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 $\Phi = [\mathtt{X}\langle \boldsymbol{a}\,;\boldsymbol{b}\rangle\ ;\ \boldsymbol{z}\ ;\ w]_h$, 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 `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 `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 T_Φ.out$[k]$.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 `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 expressions $\mathcal{E}$ with the respective arguments ctxo.α_i, ctxo.β_j. In order to make the script more readable we denote this substitution with ctxo.$\mathcal{E}$. For example if there

is a term $\texttt{after}\, t : \cdots$ 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) As we mentioned above, the script of an output representing a contract is shared among all the contracts descending from the same initial advertisement. Here we show how to construct that script when given the term $\Phi = [\mathtt{X}_0\langle \boldsymbol{a_0}\,;\boldsymbol{b_0}\rangle\ ;\ \boldsymbol{z}\ ;\ w]_h$. Let $\mathtt{X}_1, \cdots, \mathtt{X}_n$ be the list of all clauses that can be reached by recursively following every `call` operation that appears in $\mathtt{X}_0$'s definition. We will assume that each of those clauses defines the process C_i respectively.

The script starts by specifying that a transaction redeeming this output must have it as its first input. Beside giving us a way to retrace the previous steps of a contract, this avoids the problem of having a transaction that tries to continue two distinct contracts at once, "cancelling" one of them. Next, we check the name argument to see which clause is currently encoded in the transaction output, and we decide accordingly which script is going to be executed.

$$
\begin{aligned}
&\mathsf{scr}_\Phi := \mathsf{inidx} = 1 \text{ and} \\
&\text{if } (\mathsf{ctxo.name} = \mathtt{X}_0) \text{ then } \mathsf{scr}_{C_0} \text{ else} \\
&\vdots \\
&\text{if } (\mathsf{ctxo.name} = \mathtt{X}_n) \text{ then } \mathsf{scr}_{C_n} \text{ else } \mathit{false}
\end{aligned}
$$

We can now go on to describe each term scr_C. Every process C is constructed as a choice between branches $D_1 + \cdots + D_m$. To each of these branches we associate the number n_j, defined as follows: if D_j end in a `call`, then n_j is equal to the number of called clauses; if instead 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 wants to redeem C by choosing the option presented by the j-th branch. Such a transaction must also specify the value j in the argument branch of each of its outputs. For

this reason we define, as a shorthand, the condition $\mathsf{B_j}$ as

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

To handle all the branches, we let

$$\begin{aligned}&\mathsf{scr}_C \coloneqq\\ &\quad \text{if } \mathsf{B_1} \text{ then } \mathsf{scr}_{D_1} \text{ else}\\ &\quad\ \vdots \qquad \vdots \qquad \vdots\\ &\quad \text{if } \mathsf{B_m} \text{ then } \mathsf{scr}_{D_m} \text{ else } false\end{aligned}$$

Each branch D consists of a sequence of decorations ended by a `call` or a `send` operation. In order to construct scr_D, we will first focus on removing the decorations. If there is an authorization decoration, then we want the redeeming transaction's witnesses to include a signature by the authorizing participant. To enforce this, we have the following:

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

where D' does not contain any authorization decoration.

Now we consider the waiting decorations. These are both handled by using absAfterand relAfter in order to force the respective timelock to be greater than the value specified in the decoration:

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

Finally, we can describe the terminal parts of a branch: `send` and `call`.

First consider the case $D'' = \texttt{send}\ (v_1 \to \mathcal{N}_1, \cdots, v_n \to \mathcal{N}_n)$. Here, we want the k-th output of the redeeming transaction to be a deposit output owned by $\mathcal{N}_k$, and to have value v_k. From Definition 8 we know the exact structure of these outputs, so we can use the covenant verscr to force the redeeming transaction's output to have the correct script. At the same time, arglen() forces each output to have exactly three arguments (as mentioned before, these are nonce, branch, and owner). Finally we add a proposition

that checks if the outputs' values and owners are correct, and $\mathsf{scr}_{D''}$ will be equal to:

$$\begin{aligned}
&\mathsf{arglen}(\mathsf{rtxo}(1)) = 3 \text{ and } \cdots \text{ and } \mathsf{arglen}(\mathsf{rtxo}(\mathsf{n})) = 3\\
&\text{and } \mathsf{verscr}(\mathsf{versig}(\mathbf{key}(\mathsf{ctxo}.\mathsf{arg}_3), \mathsf{rtxw}.1), 1)\\
&\text{and } \mathsf{rtxo}(1).\mathsf{value} = \mathsf{ctxo}.v_1 \text{ and } \mathsf{rtxo}(1).\mathsf{owner} = \mathsf{ctxo}.\mathcal{N}_1\\
&\quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots\\
&\text{and } \mathsf{verscr}(\mathsf{versig}(\mathbf{key}(\mathsf{ctxo}.\mathsf{arg}_3), \mathsf{rtxw}.1), n)\\
&\text{and } \mathsf{rtxo}(n).\mathsf{value} = \mathsf{ctxo}.v_n \text{ and } \mathsf{rtxo}(n).\mathsf{owner} = \mathsf{ctxo}.\mathcal{N}_n
\end{aligned}$$

Lastly, we consider $D'' = \texttt{call}\ (\mathtt{X}_1\langle \boldsymbol{a^1}; \boldsymbol{b^1}\rangle \cdot, \cdots, \mathtt{X}_n\langle \boldsymbol{a^n}; \boldsymbol{b^n}\rangle \cdot)$. Here, we want a script that evaluates to true only if for all i the i-th redeeming transaction outputs encode the contract specified by $\mathtt{X}_i\langle \boldsymbol{a_i}; \boldsymbol{b_i}\rangle$, with $i = \boldsymbol{a^i}; \boldsymbol{b^i}; \boldsymbol{c^i}; \boldsymbol{d^i}$ for some choice of the parameters c^i_j and d^i_j. As mentioned, we want all contract descending from the same initial advertisement to share the same script. We enforce this condition with the recursive covenant verrec. Now, assume $\mathtt{X}_i(\boldsymbol{\alpha^i}; \boldsymbol{\beta^i})\gamma^i\delta^i = \{\mathcal{E}_i \ \texttt{if}\ p_i\}\ C_i$, and let k^i_α be the length of $\boldsymbol{\alpha^i}$ (and similarly define k^i_β, k^i_γ, k^i_δ). We will need to check that the amount of arguments is correct, and that name refers to the correct clause $\mathtt{X}_i$, and that all the "static" arguments α_j and β_j are set to the valuea a^i_j, b^i_j appearing in D''. Lastly, we need to make sure that the funding precondition of the clause is satisfied, so we check that the proposition p evaluates to true and that the value of the outputs agree with the one specified in the precondition. Notice that these two properties are checked on the arguments of the redeeming transaction's output. This leads us to $\mathsf{scr}_{D''}$ being equal to:

$$\begin{aligned}
&\mathsf{verrec}(1) \text{ and } \mathsf{arglen}(\mathsf{rtxo}(1)) = 3 + k^1_\alpha + k^1_\beta + k^1_\gamma + k^1_\delta\\
&\text{and } \mathsf{rtxo}(1).\mathsf{name} = \mathtt{X}_1 \text{ and } \mathsf{rtxo}(1).\mathsf{value} = \mathsf{rtxo}(1).\mathcal{E}_1 \text{ and } \mathsf{rtxo}(1).p_1\\
&\text{and } \mathsf{rtxo}(1).\alpha_1 = \mathsf{ctxo}.a^1_1 \cdots \text{ and } \mathsf{rtxo}(1).\alpha_{\mathsf{k}^1_\alpha} = \mathsf{ctxo}.a^1_{k^1_\alpha}\\
&\text{and } \mathsf{rtxo}(1).\beta_1 = \mathsf{ctxo}.b^1_1 \cdots \text{ and } \mathsf{rtxo}(1).\beta_{\mathsf{k}^1_\beta} = \mathsf{ctxo}.b^1_{k^1_\beta}\\
&\quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots\\
&\text{and } \mathsf{verrec}(n) \text{ and } \mathsf{arglen}(\mathsf{rtxo}(\mathsf{n})) = 3 + k^n_\alpha + k^n_\beta + k^n_\gamma + k^n_\delta\\
&\text{and } \mathsf{rtxo}(n).\mathsf{name} = \mathtt{X}_n \text{ and } \mathsf{rtxo}(n).\mathsf{value} = \mathsf{rtxo}(n).\mathcal{E}_n \text{ and } \mathsf{rtxo}(n).p_n\\
&\text{and } \mathsf{rtxo}(n).\alpha_1 = \mathsf{ctxo}.a^n_1 \cdots \text{ and } \mathsf{rtxo}(n).\alpha_{\mathsf{k}^n_\alpha} = \mathsf{ctxo}.a^n_{k^n_\alpha}\\
&\text{and } \mathsf{rtxo}(n).\beta_1 = \mathsf{ctxo}.b^n_1 \cdots \text{ and } \mathsf{rtxo}(n).\beta_{\mathsf{k}^n_\beta} = \mathsf{ctxo}.b^n_{k^n_\beta}
\end{aligned}$$

Note that the execution can carry on only if the conditions are verified for all of the redeeming outputs, mirroring the fact that symbolically all contracts are activated simultaneously.

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 **key**, which maps symbolic participants to their public key; and **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 **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; **t**, which is a list of integers with the same length of **in** that will be used to construct the relative timelocks; and **nonce**, a non empty list of integers that will be used to construct the nonce argument in each output.

Definition 9 (Compiler)**.** Below we define the function $\text{₿}_{\mathbf{adv}}$, 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

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

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 with $\mathtt{call}\ (\mathtt{X}_1\langle \boldsymbol{a_1}\,;\boldsymbol{b_1}\rangle, \cdots, \mathtt{X}_n\langle \boldsymbol{a_n}\,;\boldsymbol{b_n}\rangle)$, 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 8. 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 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 10 (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) $\mathsf{o.arg}_3 = \mathtt{X}$;

(ii) $\forall i \in \{1 \cdots k_\alpha\}.\ \mathsf{o.arg}_{3+\mathsf{i}} = a_i$;

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

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[2].

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.

[2]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.

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 **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 ran **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 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 12).

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.

4.4 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 this section we will model the choices of participants as probabilistic algorithms, which we construct by as 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 11 (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 12 (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 13 (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 14 (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 15 (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

$$R_0^c = \mathsf{T}_0 \cdots \mathsf{A}_i \to *: (K^p_{\mathsf{A}_i}(r_{\mathsf{A}_i})) \cdots \\ \cdots \mathsf{B}_j \to *: (K^p_{\mathsf{B}_j}(r_{\mathsf{Adv}})) \cdots$$

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

Definition 16 (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* $\mathcal{B}[R^c]$ as follows:

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

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

1. Its blockchain $\mathcal{B}[R^c]$ 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[3]; 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 18 (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 19 (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 $\mathsf{B} \to *: m$. However, since they do not have direct access to B's random source, they will be, with overwhelming

[3]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 $\mathcal{B}[R^c]$ 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 $\forall j.\ t - t_j \geq \delta_j$.

probability, unable to forge B's signature, since they are restricted to using a polynomial time strategy.

Definition 20 (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 $\Lambda_i^c = \Sigma_{\mathsf{A}_i}^c(\dot{R}^c, r_{\mathsf{A}_i})$, we have $\Sigma_{\mathsf{Adv}}^c(\dot{R}^c, \mathbf{\Lambda^c}, r_{\mathsf{Adv}}) = \lambda^c$.

4.5 Coherence

In this section 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 21 (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[4];
5. *prevTx* is the empty function.

[4]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

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

   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

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

   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. $\alpha = adv(\Phi)$, $\lambda^c = \mathsf{A} \to *\colon \mathsf{T}$, 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 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. $\lambda^c = \mathsf{A} \to *\colon m$ 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 1. *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 2. *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 $\mathcal{B}[R^c]$. 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 $\mathcal{B}[R^c]$. 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 21 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 [5] in cases that 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 3. *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 $\langle C, v \rangle^t_x$ is in R^s, then $txout(x)$ is the output of a compiler generated transaction. Moreover $txout(x)$ is a contract output encoding C.*

[5] Later, in Proposition 3 we will give a more profound justification on why the covenant really forces the value to be correct

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 4. *Assume that:*

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

The sum of all values stored in transaction outputs that do not belong to the image of txout (which we denote as 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 21, 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 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 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. □

4.6 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 Theorem 3 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 5. *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*

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

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 input of T. Obviously, one of these inputs will be $txout(x)$. By Theorem 3 $txout(x)$ 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, along with the fact that we know T to be a valid 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 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 $(\text{ran } txout \setminus \{txout(x)\})$, and the other is set to be the sum of the values of the inputs that are not in 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 section motivates why we are able to do so.

From Proposition 3 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

$$\begin{aligned}&\mathsf{outlen}(\mathsf{rtx}) = n_k \text{ and } \mathsf{rtxo}(1).\mathsf{branch} = k \text{ and}\\ &\cdots \text{ and } \mathsf{rtxo}(n_k).\mathsf{branch} = k.\end{aligned}$$

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 $\boldsymbol{?^i}$ 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

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

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 $\mathit{txout}(x)$. 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 $\mathit{txout}(x)$'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:

$$\begin{aligned}&\mathsf{rtxo}(i).\alpha_\mathsf{1} = \mathsf{ctxo}.a^i_1 \text{ and } \cdots \text{ and}\\ &\mathsf{rtxo}(i).\alpha_\mathsf{m} = \mathsf{ctxo}.a^i_m\end{aligned}$$

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

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

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 6. *Under the same hypotheses of Theorem 5, 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 $\mathcal{B}[R^c]$. Moreover, these outputs are unspent so, by Proposition 2 they must actually appear in the configuration. Also, thanks to Propositon 4, 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. □

4.7 Translating symbolic strategies

This section 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 18 $\aleph(\Sigma^s_{\mathsf{A}})$ 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 7. *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 to 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 18 are respected. In the following paragraphs we will show how to do that. When calling the compiler $\mathbb{B}_{\mathbf{adv}}$ 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 12 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 **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 $\mathsf{script} = \mathsf{false}$.

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 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.

4.8 Security of the compiler

In this section we prove the main result of this chapter: 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 8 (Security of the compiler)**.** *Let $\mathbf{\Sigma}^s$ be a set of computational strategies for all honest participants, and $\mathbf{\Sigma}^c$ 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*

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

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 5 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 5 also tells us that Φ is in the form $\left[\bar{D}\ ;\ \boldsymbol{z}\ ;\ w\ ;\ (x\,,j)\right]_h$ and Theorem 6 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 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 21; 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 ℵ 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. □

Chapter 5

An High-Level Language for UTXO Model

5.1 An high-level language to abstract 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 5.1 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

```
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 5.1: A crowdfund contract in HELLUM.

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 5.1). Regarding loops, we note that in general they are discouraged even in Solidity, since they may vehicle gas exhaustion attacks [106] 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 [11]. 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 in-

volve 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 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 5.2: General form of HELLUM contracts.

5.2 Compiling HeLLUM into Illum

We describe in this section how to compile high-level contracts written in HELLUM to the intermediate-level language ILLUM.

HeLLUM syntax

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 5.2.

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.

HeLLUM compilation

The first step in compiling an HELLUM contract is to transform its method in the following standardized form:

Definition 22 (Normal form of HELLUM methods)**.** The method of an HELLUM contract is in normal form when:

- expressions do not contain internal calls to pure functions (these calls are macro-expanded);

```
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 5.3: A simple contract in HELLUM.

- 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 (Figure 5.3) is displayed in Figure 5.4. 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.

Compilation: transforming a contract to normal form Here we describe the series of code transformations to bring contracts in the normal form described above. This phase is split into several steps:

1. macro-expand calls to pure functions into the corresponding expressions;

```
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 5.4: Normal form of the `Test` contract.

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 [99], 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.

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 `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 the normal form to Illum clauses Now that the contract has been reduced to a normal form, we can generate the Illum clauses associated to each function. More specifically, each function `f` splits into two Illum clauses, called `f_run` and `f_next`. Again, the full implementation can be found in https://github.com/bitbart/illum-lang/.

The clause `f_run` is used to take the parameters of `f` and run the function body. It has one branch for each of the conditional branches of `f`: these branches are enabled by the same conditional guards. They simultaneously

perform the payments and call `f_next` with the updated state passed 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 5.5, 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 5.4. 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 5.4. 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 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.

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).

```
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 5.5: Translation of the `Test` contract in ILLUM.

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.

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 5.1: Benchmark of smart contracts in HeLLUM, displaying lines of code (LoC) and size in bytes (B).

5.3 Evaluation of HeLLUM

To evaluate the practicality of Illum as a compilation target of higher-level contract languages, 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 5.1 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.

5.4 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 [104] 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* computational 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. Another way to approach the security of smart contract language has been the use of resource-oriented type system, such as the Move programming language, to give static guarantees against the mismanagement of assets. For instance [107] compiles the Move language to TEAL (the native contract language of Algorand). This allows the users to develop Algorand users to develop smart contracts while keeping the native contract language of Algorand, while preserving the safety features given by Move's linear type system. HeLLUM and Illum do not implement linear types and therefore do not have these features. However, we it could be possible to encode them by exploiting more advanced features of the eUTXO model, such as Cardano custom tokens and custom policies.

In the UTXO realm, a variety of contract languages have been proposed, starting from Bitcoin Script [38], 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 [87] and Miniscript [116]. 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 [12]. The lan-

guages [87, 116] however can only specify the individual transactions used in these protocols, and not the overall global contract. BitML [32] 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 the crowdfunding in Section 5.1, as well as auction contracts, which allow for an unbounded number of steps. The work [23] 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 [100].

To overcome the expressiveness limitations of the Bitcoin UTXO model, the Cardano blockchain extends it with some additional functionalities [43, 46]: (i) special transaction fields to store contract state; (ii) a mechanism to preserve contract code along chains of transactions; (iii) native custom tokens [45]; (iv) an expressive scripting language [93]. 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 4.1. 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 3.2. 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 [4], which stores in the script 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, verscr(h, i) would simply check that the hash in rtxo(i).script is equal to h. Similarly, the verrec(i) covenant would check that the hash in rtxo(i).script is equal to the hash in the current script, ctxo.script. Both checks can be done very efficiently, as one just needs to compare two hashes. A further optimization can be achieved by exploiting Taproot [90], 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 [9, 56, 57, 102], while some others are based on the static analysis of contracts [22, 92]. In particular, [56] 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.

Chapter 6

Scalable UTXO Smart Contracts via Fine-Grained Distributed State

6.1 Introduction

Current smart contract platforms can be roughly partitioned in two main classes: *account-based* and *UTXO-based*. In an account-based blockchain, transactions update the state of a contract by specifying which method to execute; the actual contract state and balance are not stored on-chain, but rather reconstructed off-chain by nodes that execute the invoked methods. Account-based blockchains can be further classified in *stateful* or *stateless*, depending on whether the contract state is associated to a single (contract) account, or it is scattered across multiple accounts. While in account-based platforms (of either kind) the contract state is stored off-chain, in UTXO-based ones it is recorded directly *on-chain* within transactions, which also carry the contract logic. The impact of this on-chain overhead depends on the actual platform: for instance, in Bitcoin it is negligible — but just because Bitcoin supports limited smart contract functionality, where the size of the contract state is severely constrained [12]. To overcome this limitation, Cardano introduced an *extended* UTXO model (eUTXO [43, 46, 74]), revisiting the structure of transactions so to allow contracts to read and write much larger (on-chain) states, and extend the script language to be Turing-powerful. In the eUTXO model, the impact of the above mentioned on-chain overhead is no longer negligible, as invoking a contract method amounts to publishing on the blockchain a transaction that includes the *entire* new contract state and maintains the old contract logic.

The account-based and UTXO-based paradigms have well-known differences in the programming style of their smart contracts [17,41]. Furthermore, their fundamentally diverse nature deeply affects the performance (e.g., the transaction throughput), costs and scalability of smart contracts. A well-known weakness of the stateful account-based model, which is the one followed by the most popular smart contract platforms such as Ethereum, is that it hinders the parallelization of validator nodes [60]. Indeed, detecting when two transactions are parallelizable is a hard problem, and approaches that try to improve the node performance require complex techniques based on optimistic execution [5,9,56,61,120] or static analysis [22,92]. Not being able to efficiently use the computational resources of validators is a problem, since validators perform the vast majority of work in contract executions. Parallel validation is instead a significant selling point of UTXO-based blockchains. There, detecting when two transactions are parallelizable is trivial: just check that their inputs satisfy the Bernstein conditions [22]. In other words, when

one transaction spends an output, we need to ensure that the same output must not be accessed (either spent or read) by the other transaction. However, in smart contract platforms based on the UTXO model, like Cardano, smart contracts are typically implemented as a *single* transaction output, encoding both the contract logic and the current contract state. In this way, however, actions on the same contract are *never* parallelizable, since they attempt to spend the same output. Another problem is that any transactions acting on a contract must include the *whole* updated contract state. When this gets large (e.g., for contracts storing user data in maps), it becomes a performance bottleneck. Besides that, transaction size is usually capped (e.g., Cardano has a 16KB hard cap), so one can easily reach a point when further contract updates are forbidden (this could also be exploited by DoS attacks). Although the blockchain size could be reduced by storing on-chain only the hash of the contract state [70], validators still need access to the entire state corresponding to such hashes, which still requires a significant amount of network communications. In summary, smart contracts in the UTXO model currently have poor scalability when the contract state becomes large.

6.1.1 Distributing the contract state

In this chapter we propose a framework to improve the performance and scalability of UTXO-based contracts. To this purpose, we introduce a hybrid UTXO model, which we name hUTXO, combining features from both UTXO and account-based models. The key idea is to *distribute the contract state over multiple UTXOs*. As a basic example, if the contract state contains two variables, we store their values in distinct UTXOs: in this way, contract actions accessing different variables can be parallelized. More in general, we distribute arbitrary contract states, including those comprising dynamic data structures such as key-value maps. This is done in a fine-grained way: e.g., maps are distributed storing each key-value association $[k \mapsto v]$ in its own UTXO. While distributing the contract state has a clear advantage in parallelization, maintaining the *contract balance* in such distributed approach is problematic. Of course, even if the state is distributed, we still could store the contract balance in a single UTXO, but this would introduce a single point of synchronization, hindering parallelization. Alternatively, we could split the balance across multiple UTXOs, but this would not completely solve the problem. For instance, consider a contract with a balance of 2 tokens, split in two distinct UTXOs. If we want to perform two actions, each withdrawing 1 token from the contract, it may happen that both of them attempt to spend the same UTXO, hence causing a conflict, hindering parallelization again. This problem can manifest itself more frequently when a withdrawal requires spending several UTXOs of lesser value; indeed, conflicts — which arise when two actions attempt to spend the same output — become more likely when the actions need to consume many UTXOs. To address this problem, in our hUTXO model we separate the contract state (stored in UTXOs) from the contract balance (stored in a stateless account). In this way, withdrawing from the contract (i.e. from its account) never causes conflicts, provided the balance is sufficient.

6.1.2 hUTXO: basic characteristics

While distributing the contract state and isolating the balance enables the parallel validation of transactions, this additional complexity can expose contracts to new kinds of attacks. Recall that in UTXO-based models, adversaries can always create a new UTXO with arbitrary fields (including the token amount, as long as they spend the needed tokens). This poses no issue

to centralized contracts that involve a single UTXO carrying both the data and the contract balance: indeed, the behaviour of that UTXO is unaffected by the presence of other UTXOs the adversary could create. By contrast, in distributed contracts, where the contract state is encoded in multiple UTXOs with a small monetary value, an adversary can cheaply forge UTXOs so to corrupt the contract state.

To thwart forgery attacks, our hUTXO model proposes a novel combination of techniques:

- We introduce *contract IDs*: the same ID is shared by all the UTXOs contributing to the contract state and by the associated balance account.

- hUTXO validators ensure that new transactions can spawn new UTXOs with pre-existing contract IDs *only* when they are vetted by a script of a previous UTXO already linked to that ID. In this way, only descendants of a contract-related UTXO can claim to represent a part of the contract state.

- Finally, we use scripts to ensure that the new UTXOs encode a new state as mandated by the contract logic.

6.1.3 hURF: A contract language for hUTXO

The last item above highlights a possible issue: since programming contract scripts of UTXO transactions is notoriously error-prone [17], developers must be extremely careful not to introduce vulnerabilities in the transaction scripts. To address this issue, we propose a high-level contract language, named tchURF (for hUTXO Rule Format), and a compiler from hURF to hUTXO transactions. hURF abstracts from the hUTXO transaction structure, modelling contracts as sets of rules reading and updating the contract state. A contract rule can verify a precondition on the contract state, update the state by performing a sequence of variable/map assignments, and transfer currency from/to users. The hURF compiler guarantees that the scripts in the hUTXO transactions resulting from compilation enable *exactly* the state/balance updates specified by the hURF rules. Overall, hURF relieves developers from worrying about hUTXO-level attacks, so that they can instead focus on the (high-level) contract logic.

Contributions In summary, in this section we will explore the following contributions:

- A new hybrid blockchain model, hUTXO, which extends the UTXO model with Cardano-like contract states, contract IDs, and contract balance accounts. The transaction validation conditions of hUTXO support the secure distribution of contract states
- A high-level contract language, hURF, with a secure compiler to hUTXO
- A parallel algorithm for validating transactions blocks in the hUTXO model
- A prototype implementation of a hUTXO validator, featuring both sequential and parallel validation, and an evaluation on a benchmark of contracts inspired by real use cases like crowdfund, registry and multisig wallet. Our results demonstrate that the parallel validator consistently achieves a speedup approaching the number of available threads. Furthermore, the distribution of the contract state results in a time and space complexity that grows linearly with the size of the state. This is coherent with our expectations, since in the centralized-state contract, the larger the state, the more time it takes to duplicate it in the redeeming transaction. Notably, the size of transactions in the distributed-state contract is constant, making it possible to execute contracts with huge dynamic state even in blockchains where the size of transactions is strictly limited (such as, for instance, in Cardano).
- To foster the reproducibility of our experiments, we have made our simulator and benchmarks publicly available as a Docker image on Zenodo [29].

6.2 From eUTXO to hUTXO

6.2.1 A crowdfund contract in eUTXO

To illustrate our technique, we consider a crowdfund contract with the following workflow. First, donors can deposit any amount of tokens in the contract. Then, after a first deadline, the contract owner can withdraw all the deposited tokens, provided that the total amount is greater than the required goal. After a second deadline, the donors can take their donations back, if they are still within the contract.

This contract can be implemented in the eUTXO model by storing the whole map of donations in the *datum* field of a single transaction output o. Using this approach, making a donation amounts to spending the output o and creating a new one o' whose *datum* has the same map as o except for the donor entry, which is increased by the donation amount. After the first deadline, the contract owner has a chance to withdraw the funds, if there are enough, by spending the last output. After the second deadline, if the funds are still in the contract, each donor can take their part back, according to the value stored in the map. Doing so requires spending the last output o and creating a new one o' that has the same *datum* map except that the entry for the donor has been removed.

This contract implementation is, in fact, *centralized*: the whole contract state is stored in a single output. This design choice has three main downsides:

- Each donation transaction contains a copy of the old contract state with small changes. With N distinct donors, the state map becomes larger and larger, requiring $O(N^2)$ total blockchain space. This is overly inefficient.
- When multiple donors attempt to deposit at the same time, they all have to spend the same output o — leading to *UTXO congestion* [68]. Since the eUTXO model prevents double spending, all these operations are in conflict and race against each other. Consequently, only one donation will succeed, forcing the other donors to retry with a new transaction.
- This makes state updates inherently *sequential*, as they rely on a chain of transactions redeeming each others' outputs. This hinders the parallel validation of transactions.

6.2.2 Distributing the contract state

The key idea to overcome these issues is to distribute the contract state across multiple UTXOs. For example, in our crowdfund contract, the state is a map associating participant addresses p_i to their donated amount $a_i \neq 0$. We represent it as:

$$[p_1 \mapsto a_1, \ldots, p_n \mapsto a_n] \tag{6.1}$$

The participants not occurring in the map representation (6.1) are implicitly associated with the default value *zero*. To distribute the map, we exploit the standard total ordering of addresses. For simplicity, suppose $p_{\mathsf{min}} < p_1 < \cdots < p_n < p_{\mathsf{max}}$, where p_{min} and p_{max} correspond to the minimum and maximum addresses, respectively. Consequently, we can now denote the same state map (6.1) using the alternative representation:

$$(p_{\mathsf{min}}, p_1)[p_1 \mapsto a_1](p_1, p_2)[p_2 \mapsto a_2](p_2, p_3) \ \ldots \\ \ldots (p_{n-1}, p_n)[p_n \mapsto a_n](p_n, p_{\mathsf{max}}) \tag{6.2}$$

In the new representation (6.2), an item $[p_i \mapsto a_i]$ denotes that address p_i is associated with $a_i \neq 0$, whereas an item (p_j, p_k) denotes that all addresses in that *open interval* are associated to the default value zero.

To distribute the state, we store each item in its own output. For instance, item $[p_2 \mapsto a_2]$ is stored in the *datum* field of an output o_{p_2}, while item (p_2, p_3) is stored in the *datum* field of another output o_{p_2,p_3}.

This state distribution makes operating on the contract more complex. If participant p_i has already donated a_i tokens, they can donate additional b_i tokens by consuming the output o_{p_i} (representing $[p_i \mapsto a_i]$) and creating a new output o'_{p_i} (representing $[p_i \mapsto a_i + b_i]$). If, instead, p_i has not donated yet, they can donate b_i tokens by *splitting* an open interval. This requires p_i to find the open interval $(p_j, p_k) \ni p_i$ represented by output o_{p_j,p_k}, consume it, and create *three* new outputs $\mathsf{o}'_{p_j,p_i}\mathsf{o}'_{p_i}\mathsf{o}'_{p_i,p_k}$ to represent the items $(p_j, p_i)[p_i \mapsto b_i](p_i, p_k)$. Similarly, to take their donation back after the second deadline, p_i must *merge* three items into a single interval item. This requires p_i to find the three outputs $\mathsf{o}_{p_j,p_i}\mathsf{o}_{p_i}\mathsf{o}_{p_i,p_k}$ representing the items $(p_j, p_i)[p_i \mapsto a_i](p_i, p_k)$, spend them, and create a new output o_{p_j,p_k} for the new open interval (p_j, p_k). This effectively updates the map so that p_i is now associated to the default value zero.

This distributed implementation of the contract solves all the issues of the centralized one discussed before, in that:

- The *datum* fields now contain a bounded amount of data (one item, $O(1)$), and each operation consumes and creates at most three outputs. Hence, performing N donations only requires $O(N)$ space instead of $O(N^2)$ as in the centralized implementation.
- When multiple participants attempt to donate at the same time, only those operations falling in the same interval cause conflicts. After a few donations, the contract state becomes fragmented in several intervals, gradually reducing the conflict probability.
- Fewer conflicts result in more transactions with disjoint outputs, creating greater opportunities for parallelization.

Handling more complex states In general, the state of a contract does not involve a single map as shown in the previous example, but can comprise multiple maps and variables.

However, the general case can be reduced to the specific single-map case, by a simple *flattening* of the state. For instance, in order to represent maps `m1`, `m2` and variables `x1`, `x2`, we can use

$$\begin{aligned}[\,&\texttt{hash("map\_m1[12]")} \mapsto v_1,\ \ldots,\ \texttt{hash("map\_m1[45]")} \mapsto v_2,\\ &\texttt{hash("map\_m2[54]")} \mapsto v_3,\ \ldots,\ \texttt{hash("map\_m2[89]")} \mapsto v_4,\\ &\texttt{hash("var\_x1")} \mapsto v_5,\ \texttt{hash("var\_x2")} \mapsto v_6\,]\end{aligned}$$

where `hash` is a collision-resistant hash function.

Using hashes as done above ensures the global map keys have constant size. It also helps in spreading the keys in an almost-uniform way over the key space.

6.2.3 Handling tokens

In the previous discussion of the distributed approach we did not mention how to store the contract balance. While in the centralized approach there is only a natural choice — storing all the tokens in the single contract output — in the distributed contract there is no straightforward solution.

One option is to continue storing the whole balance in a single "balance output", even in the distributed case. This, however, would re-introduce conflicts between any pair of operations affecting the contract balance, since they would need to spend the same balance output. Since this would negate

many of the benefits of the distributed approach, we must look for other options.

Another option is to use multiple "balance outputs", spreading the balance over multiple UTXOs. This would reduce conflicts since operations spending disjoint outputs can now be executed in parallel. Still, this solution is not ideal. For instance, when transferring 100 tokens from a contract, one has to consume one or more balance outputs to reach such amount. Depending on the actual fragmentation of the tokens over balance outputs, it might be impossible to reach the exact amount of 100 tokens, forcing the operation to consume more and then put the exceeding tokens back in the contract by creating a new balance output. This can increase conflicts, as more tokens need to be consumed than the operation actually requires. Another drawback is that one must consume a variable amount of balance outputs, making the contract logic more complex. Finally, since each transaction attempts to consume many balance outputs, it becomes more unlikely for different transactions in the mempool to have disjoint sets of inputs, so increasing the probability of conflicts. We stress that a single shared input would be enough to cause a conflict between two transactions.

Our hybrid accounting model hUTXO avoids these issues by distributing the contract data across several UTXOs, while keeping the contract balance into a *contract account*, outside of the UTXOs. By storing the contract balance in a separate account, we effectively make the contract tokens indistinguishable from each other: when transferring 100 tokens from the contract we no longer have to specify *which* 100 tokens (or more) to consume, as we must do in the regular UTXO model. Using our approach, we will never have to consume more than 100 tokens and can always run a token transfer in parallel with others, provided there is enough balance in the contract.

Crucially, hUTXO contract accounts only contain tokens, unlike stateful account-based blockchains such as Ethereum, where contract accounts also store the contract state and code. We remark that in hUTXO, tokens can be kept both in contract accounts and in UTXOs, while contract data is only stored in the *datum* fields of transaction outputs, as in the eUTXO model. When an output contains distributed contract data, we will not store tokens inside it, preferring instead to deposit them in the contract account.

6.2.4 Transaction validity in hUTXO

Despite the advantages that the distributed approach offers over the centralized one, there is a drawback: in UTXO-based models, the adversary can freely create outputs with arbitrary *datum* fields, and this makes distributed contracts potentially vulnerable to attacks that are not present in the centralized approach.

To illustrate an attack of this kind, consider the crowdfund contract in a state where the adversary p_A has donated 1 token. This state is rendered by items $\cdots(p_j, p_A)[p_A \mapsto 1](p_A, p_k)\cdots$ and outputs $\cdots \mathsf{o}_{p_j,p_A}\mathsf{o}_{p_A}\mathsf{o}_{p_A,p_k}\cdots$ Now, the adversary can forge an output $\hat{\mathsf{o}}_{p_A}$ whose *datum* field denotes the item $[p_A \mapsto 100]$. Note that $\hat{\mathsf{o}}_{p_A}$ (like o_{p_A}) does not hold tokens but only data, so the forgery costs the adversary nothing (other than transaction fees). By exploiting this forgery, once the deadlines have passed the adversary can perform a refund operation by spending two contract outputs and their own forged output, as in $\mathsf{o}_{p_j,p_A}\hat{\mathsf{o}}_{p_A}\mathsf{o}_{p_A,p_k}$, and withdraw in this way 100 tokens from the contract balance. This is a clear attack, as it allows the adversary to receive more tokens than they donated.

The transaction validation of the eUTXO model [46] does not offer direct countermeasures against this kind of attacks.[1] Indeed, eUTXO scripts can inspect the fields of sibling outputs, i.e., outputs that are referred to as inputs in the redeeming transaction. However, the fields of the forged output $\hat{\mathsf{o}}_{p_A}$ are *indistinguishable* from those which would appear in a legit output if p_A had donated 100 tokens. Because of this, the scripts of the siblings $\mathsf{o}_{p_j,p_A}\mathsf{o}_{p_A,p_k}$ can not detect the forgery and consequently stop the attack. Technically, in order to detect the forgery, scripts would need to inspect the ancestors of $\hat{\mathsf{o}}_{p_A}$: doing so, they would observe that $\hat{\mathsf{o}}_{p_A}$ does not descend from the transaction that originally deployed the contract. Since inspecting such ancestry would be clearly inefficient, in our hUTXO model we introduce an alternative solution: *contract IDs*. Back to our example, we assign to the distributed crowdfund contract a unique ID, which is then used to mark legit UTXOs, i.e. those storing valid data for that contract. When considering a transaction for inclusion in the blockchain, hUTXO validators enforce conditions that prevent the adversary from forging UTXOs with the same ID. Intuitively, validators only accept a transaction T if it falls in one of the following cases:

[1] Some mitigation, however, was proposed in the literature, see Section 6.9.

- T deploys a new contract, creating outputs with a *fresh* contract ID. In this way, the adversary can still create an output $\hat{\mathsf{o}}_{p_A}$ with a forged *datum* field, but only with a distinct contract ID. This prevents the attack, since when redeeming $\mathsf{o}_{p_j,p_A}\hat{\mathsf{o}}_{p_A}\mathsf{o}_{p_A,p_k}$ the scripts of the two legit outputs can detect that $\hat{\mathsf{o}}_{p_A}$ does not belong to the same contract, and reject the transaction.

- T performs an operation on an existing contract, spawning new outputs with that contract ID, but only when some input of T refers to a pre-existing UTXO with the same contract ID. In this way, the adversary is allowed to create an output $\hat{\mathsf{o}}_{p_A}$ having the same contract ID of the distributed crowdfund contract. However, its *datum* field must be vetted by the script of the pre-existing contract UTXO. Such contract script can be designed so to accept those *datum*s that respect the contract logic. Doing so thwarts the attack, since the adversary can increase the amount of donated tokens in the donations map only if they are actually paying them.

- T performs an operation not involving contract IDs in any way (e.g., simple token transfers). In this way, the adversary can create an output $\hat{\mathsf{o}}_{p_A}$ with any *datum* field, but only with no associated contract ID. As in the first case, such $\hat{\mathsf{o}}_{p_A}$ cannot interfere with the contract, preventing the attack.

6.3 The hUTXO blockchain model

In this section we provide a more technical description about our hUTXO model. This model is similar to Cardano's eUTXO model, in that a transaction output contains a *datum* field which can be used to store contract data. As in the eUTXO model, *covenants* [24, 83, 88] are used to ensure that the *datum* is updated according to the smart contract behaviour. We further extend the eUTXO model by adding a *ctrId* field to the transactions, which represents the ID of the contract which is being affected by the transaction. If no contract is being involved, *ctrId* is set to zero. Our transaction outputs are then augmented with a boolean *inContract* field, marking which outputs belong to the contract specified by *ctrId*.

Finally, we model the balance of contracts as a map form contract IDs to currency. This map is not stored inside the UTXOs nor inside the transactions. Instead, it is a part of the implicit blockchain state, as it happens in the account-based model followed by e.g. Ethereum. This makes our blockchain model a hybrid UTXO-account one.

6.3.1 Structure of hUTXO transactions

In our model, we will use data of the primitive types:

$$\mathsf{Bool}, \mathbb{N}, \mathsf{PubK}, \mathsf{Hash}, \mathsf{TokenId}, \mathsf{Script}, \mathsf{Data}$$

The meaning of most of these data types is straightforward. We only mention that Script is the type of the redeeming scripts, while Data is a generic type that can hold arbitrary data (concretely represented as a bitstring).

Our model is formalized in Figure 6.1. There, we describe the format of Inputs, Outputs, and transactions (Tx). Below, we provide some intuition about the fields of a transaction $\mathsf{T} : \mathsf{Tx}$.

- Outputs store tokens (denoted by field *value*: Value) and data (denoted by field *datum*: Data). Each output has a *validator* script, specifying the condition under which it can be either spent or checked by another transaction. The *inContract* flag determines whether the output and its data are part of a contract or not.

- Every input references a previous unspent output. The *spent* flag determines whether the output is *spent* (i.e., T grabs its tokens) or to

$$\mathsf{TxId} \stackrel{\text{def}}{=} \mathsf{Hash}$$

$$\mathsf{CtrId} \stackrel{\text{def}}{=} \mathsf{Hash}$$

$$\mathsf{Wallet} \stackrel{\text{def}}{=} \mathsf{TokenId} \to \mathbb{N}$$

$$\mathsf{OutputRef} \stackrel{\text{def}}{=} (\mathit{txId}\colon \mathsf{TxId}, \mathit{index}\colon \mathbb{N})$$

$$\mathsf{TimeInterval} \stackrel{\text{def}}{=} (\mathit{from}\colon \mathbb{N}, \mathit{to}\colon \mathbb{N})$$

$$\mathsf{Output} \stackrel{\text{def}}{=} (\mathit{value}\colon \mathsf{Value},\ \mathit{validator}\colon \mathsf{Script},\ \mathit{datum}\colon \mathsf{Data}, \mathit{inContract}\colon \mathsf{Bool})$$

$$\mathsf{Input} \stackrel{\text{def}}{=} (\mathit{outRef}\colon \mathsf{OutputRef}, \mathit{redeemer}\colon \mathsf{Data}, \mathit{spent}\colon \mathsf{Bool})$$

$$\mathsf{Tx} \stackrel{\text{def}}{=} (\mathit{in}\colon \mathsf{List[Input]}, \mathit{out}\colon \mathsf{List[Output]}, \mathit{signers}\colon \mathsf{List[PubK]}, \mathit{validityTime}\colon \mathsf{TimeInterval}, \mathit{fee}\colon \mathbb{N}, \mathit{ctrId}\colon \mathsf{CtrId})$$

$$\mathsf{Accounts} \stackrel{\text{def}}{=} \mathsf{CtrId} \to \mathsf{Value}$$

$$\mathsf{Ledger} \stackrel{\text{def}}{=} (\mathit{txs}\colon \mathsf{List[Tx]}, \mathit{accounts}\colon \mathsf{Accounts}, \mathit{time}\colon \mathbb{N})$$

Figure 6.1: Types for the hUTXO model.

checked (i.e., T just reads its fields, including the *datum*, leaving it unspent[2]). The output script must be satisfied regardless of the *spent* flag. An input also contains a *redeemer*: a piece of data that can be checked by the referred output script.

- The *validityTime* field specifies time constraints for the validation of the transaction.

- The *fee* field specifies the amount of native cryptocurrency that the transaction pays to validators.

- The *ctrId* field uniquely identifies a contract in the blockchain. More specifically, an output o with $\mathsf{o}.\mathit{inContract} = \mathit{true}$ in T is considered part of the contract specified by $\mathsf{T}.\mathit{ctrId}$. Indeed, the current contract

[2]The ability of reading output data without spending the output is similar to Cardano's *checked inputs* [71].

state is collectively represented by all such unspent outputs, distributed across all the transactions with that *ctrId*.

The Ledger state comprises the list of the transactions that have been appended so far (*txs*), the contract balances (*accounts*), and the *time*, abstractly represented as a natural. The *accounts* field is a mapping that associates to each contract ID the amount of tokens it owns. Since the balance of the contracts is represented in the *accounts* map, there is no need to also store their tokens in the UTXOs. For this reason, the outputs having $inContract = true$ usually carry a *value* of zero tokens. Note that such value-less outputs can still be useful, since they can hold a part of the contract state in their *datum*, as well as a *validator* script to implement the contract behaviour.

6.3.2 Validity of hUTXO transactions

We list the conditions needed for the validity of a transaction $\mathsf{T} : \mathsf{Tx}$ in a given $(\mathcal{B}, \mathcal{A}, t) : \mathsf{Ledger}$. As an auxiliary notion, we say that an output is *unspent* in $\mathcal{B}$ if there is no transaction T in $\mathcal{B}$ that has an input referring to that output with $spent = true$. Then, we say that a transaction T is valid when:

1. All inputs of T refer to unspent outputs in $\mathcal{B}$.

2. The inputs of T with $spent = true$ refer to distinct outputs in $\mathcal{B}$. Note instead that the inputs of T with $spent = false$ may refer to the same output of any other input.

3. At least one input of T has $spent = true$.

4. The current time t falls within T's *validityTime* interval.

5. For each input of T (including those with $spent = false$), if it refers to the output o then evaluating $\mathsf{o}.validator$ yields $true$.

6. For each input of T, if it refers to some output o in T' with $\mathsf{o}.inContract = true$, then $\mathsf{T}'.ctrId = \mathsf{T}.ctrId$.

7. If all inputs of T refer to outputs with $inContract = false$, then:

   (a) If all outputs of T have $inContract = false$, then $\mathsf{T}.ctrId = 0$.

   (b) Otherwise, if some output of T has $inContract = true$, then $\mathsf{T}.ctrId$ is equal to the hash of the $outRef$ of the first input of T with $spent = true$.

8. If $\mathsf{T}.ctrId = 0$, let v_{in} be the sum of all the $\mathsf{o}.value$ where o is referred by an input of T with $spent = true$. Moreover, let v_{out} be equal to the sum of all the $\mathsf{o}'.value$ of outputs o' of T, and v_{fee} be a value of $\mathsf{T}.fee$ units of the native cryptocurrency. We require that $v_{in} \geq v_{out} + v_{fee}$.

9. If $\mathsf{T}.ctrId \neq 0$, then let $bal = \mathcal{A}(\mathsf{T}.ctrId)$ be the current contract balance, and let v_{in}, v_{out}, v_{fee} as in Item 8. We require that $v_{in} + bal \geq v_{out} + v_{fee}$.

We now discuss the above validity conditions. Item 1 requires that the inputs have not already been spent by a previous transaction. Item 2 prevents T from double spending the same output. Item 3 requires T to spend at least one previous output. In this way, T can be appended at most once to the ledger. Item 4 ensures that the time constraints are respected. Item 5 requires that T satisfies each script $\mathsf{o}.validator$ of each output o referred to in the inputs of T. Items 6 and 7 control the creation of new contract outputs, so that they can not be forged. More specifically, Item 6 restricts the outputs that can be created inside an existing contract $ctrId$, by requiring that some input refers to a previous output o in the same contract. This makes it possible for $\mathsf{o}.validator$ to impose its own requirements on the new outputs, rejecting output forgeries by allowing only those allowed by the contract logic. By contrast, Item 7 handles the case where a fresh contract is deployed: this happens when there are no contract inputs but some contract outputs. Here $\mathsf{T}.ctrId$ is mandated to be a fresh contract ID, so that one can not maliciously reuse a previous ID and forge outputs inside such contract. Items 8 and 9 ensure that the currency provided by the inputs is enough to cover for all the outputs and the fees. If some contract is involved by the transaction, then the contract balance can also contribute to pay outputs and fees.

6.3.3 Updating the ledger

Appending a valid transaction T makes the ledger state $(\mathcal{B}, \mathcal{A}, t)$ evolve into a new state $(\mathcal{B}\mathsf{T}, \mathcal{A}', t')$. The accounts map is unchanged if T is not a contract

action (i.e., $\mathsf{T}.ctrId = 0$), while it is updated at point $\gamma = \mathsf{T}.ctrId$ otherwise:

$$\mathcal{A}'(\gamma) = \begin{cases} \mathcal{A}(\gamma) + v_{in} - v_{out} - v_{fee} & \text{if } \gamma = \mathsf{T}.ctrId \neq 0 \\ \mathcal{A}(\gamma) & \text{otherwise} \end{cases} \tag{6.3}$$

where v_{in}, v_{out}, v_{fee} are defined as in Item 8 of Section 6.3.2. The time update is non-deterministic, with the only constraint that $t' \geq t$. We do not specify further how time is handled in the model: it could be either tied to real time or to the block height of the ledger. Our model is independent from this choice.

6.3.4 Scripts

For our purposes the actual script language is immaterial, so hereafter we do not explicitly provide its syntax and semantics. Given a transaction T referring to an output o in its inputs, we just assume that the script $\mathsf{o}.validator$ can read:

(a) all the fields of T;

(b) all the fields of the *sibling* outputs, i.e. those referred to by the inputs of T (including o itself).

In particular, by item (a) it follows that the script can access the *redeemer* field in the input of T referring to o. This is analogous to the way witnesses are used in Bitcoin. Moreover, the script can access both o (by item (b)) and each output o' of T (by item (a)). This makes it possible, for instance, to enforce a covenant by checking that $\mathsf{o}'.validator = \mathsf{o}.validator$ and that $\mathsf{o}'.datum$ is related to $\mathsf{o}.datum$ according to the specific covenant logic one wants to implement. Furthermore, by item (b), $\mathsf{o}.validator$ can allow T to spend o only if T simultaneously spends a sibling o'' of a wanted form. More specifically, o'' can be required to be a contract output by checking $\mathsf{o}''.inContract$.

6.4 The hURF contract language

Although the transactions validation mechanism outlined in Section 6.3 allows one to define programmatic exchanges of tokens, it does not provide a sharp notion of smart contract. This is because the above-mentioned programmatic exchanges can result from the interaction of multiple different scripts distributed among several transactions: in this sense, the smart contract could be seen as an emerging high-level behaviour of the low-level behaviour of these transactions.

In this section we provide a higher-level model of smart contracts, hURF (hUTXO Rule Format), where the programmatic exchange of tokens is specified by a set of rules. In this way, hURF provides a lower bound to the expressiveness of hUTXO contracts, as any hURF contract can be effectively implemented in hUTXO.

Contracts as rules Syntactically, a hURF contract comprises:

- a declaration of the variables and maps that contribute to its *state* (σ), possibly including their initial values;
- a set of *rules* ($\mathbf{R}$) that specify how the contract state σ and *balance* w can be updated.

Users execute contracts by repeatedly choosing and firing contract rules. Each rule R consists of three parts:

- a *signature* providing the rule name and the sequence of formal parameters to be instantiated by users upon firing the rule;
- a set of *preconditions* that must be met in order for the rule to be fired;
- an *effect* that can modify the contract state and transfer tokens to users, affecting the contract balance accordingly.

To formalize the contract state, we first define the *base values* BVal as

$$\mathsf{BVal} \stackrel{\text{def}}{=} \mathsf{Bool} \mid \mathbb{Z} \mid \mathsf{String}$$

Then, we make state σ associate each variable $\mathtt{x}$ to a base value $\sigma(\mathtt{x}) \colon \mathsf{BVal}$, and each map $\mathtt{m}$ to a function from tuples of base values to a single base value, i.e.:

$$\sigma(\mathtt{m}) \colon \mathsf{BVal}^* \to \mathsf{BVal}$$

To denote the function $\sigma(\mathtt{m})$ we will write

$$\sigma(\mathtt{m}) = [(l_1^1, \cdots, l_n^1) \mapsto v_1] \cdots [(l_1^k, \cdots, l_n^k) \mapsto v_k] \tag{6.4}$$

for the function that takes the value v_i at the point $(l_1^i, \cdots, l_n^i)$, and the *default value* 0 at any other point.

Coherently with our state representation in (6.4), accessing the state of a map requires one to specify the map name `m` and the point $(l_1, \ldots, l_n)$ where it is accessed. We will refer to the pair $(\mathtt{m}, (l_1 \ldots, l_n))$ as a *map location*.

The contract balance w is a Wallet (i.e., a map from token types to non-negative integers) that represents the amount of tokens owned by the contract.

Expressions Expressions `e` appearing in preconditions and effects have the following form: (i) boolean, integer, and string constants; (ii) arithmetic or boolean operators; (iii) conditional expressions `if e0 then e1 else e2`; (iv) `m[e1,...,ek]`, accessing the map `m` at point `(e1,...,ek)`; (v) `hash(e1,...,ek)`, hashing a tuple of values; (vi) `e1 @ e2`, concatenating two strings; (vii) `len(e)`, the length of a string; $(viii)$ `substr(e, e1, e2)`, taking the substring from character in position `e1` to the one in position `e2` of a string `e`; (ix) `toStr(e1,...,ek)`, converting a tuple of values to a string; (x) `signedBy(e)`, checking whether user `e` authorizes the rule invocation; (xi) `validFrom`, `validTo`, the validity temporal bounds for the rule invocation; (xii) rule parameters and state variables.

Note that the language of expressions has no operators to iterate over maps. This ensures that the evaluation time of expressions does not depend on the size of maps. W.l.o.g. there are no expressions to obtain the contract balance: if needed, the developer can record in contract variables the balances of the different tokens stored by the contract. However, doing so may affect the parallelizability of contracts, so these variables must be dealt with carefully.

Preconditions A rule precondition is a list of requirements of the following forms:

- zero or more `receive(e:T)`, enabling the rule only if some user sends exactly `e` tokens of type `T` to the contract account.
- one `require(e)`, enabling the rule only if `e` is `true`.

Effects The effect of a rule allows to simultaneously assign values to contract variables and maps, while also transferring assets from the contract balance to a given user.

The effect, denoted as `S1 | ... | SN`, simultaneously executes statements `Si` of the following forms:

- `x = e`, assigning a value to variable `x`;
- `m[e1,...,ek] = e'`, updating the map `m` at `(e1,...,ek)`;
- `e.send(e':T)`, sending `e'` tokens of type `T` from the contract balance to a user (whose public key is) `e`. If the contract balance is insufficient, the rule cannot be fired.

We stress that the execution of statements is simultaneous: all the expressions in the effect are first evaluated in the *old* state, which is then updated. To ensure that such update is well-defined, we postulate that in each rule, each variable and map location is assigned at most once. More specifically, we postulate that effects satisfy the following conditions. If the effect comprises the variable assignments `x = e` and `x = e'`, then `e` and `e'` must be equal. If the effect comprises the map assignments `m[e1,...,ek] = e` and `m[f1,...,fk] = e'` with `e` different from `e'`, then the rule `require` precondition must check that `not (e1==f1 && ... && ek==fk)`. Hereafter, for simplicity we will not explicitly include this check in our rules.

Contract behaviour To describe the behaviour of a contract, besides its state and balance we must also consider the tokens held by users. These are needed to provide inputs to the contract (to fulfill its `receive` conditions), and to transfer tokens from the contract to users (through the `send` statement). We denote by $\langle \mathsf{A}, w \rangle_x$ the fact that a user A holds a *deposit* w of tokens. The name x is used to discriminate between different deposits of the same user. For instance, in the same system configuration we can have both $\langle \mathsf{A}, [1\colon \mathsf{T}] \rangle_x$ and $\langle \mathsf{A}, [2\colon \mathsf{T}] \rangle_y$ for two distinct deposits of $1\colon \mathsf{T}$ and $2\colon \mathsf{T}$ both owned by A. Invoking a rule spends deposits, burning their name, while executing `send` creates new deposits with fresh names.

As usual, several contracts might be simultaneously active, possibly instantiating the same set of rules multiple times. To account for that, we model a *contract instance* as $\langle \mathbf{R}, \sigma, w \rangle_\gamma$ where $\mathbf{R}$ is the set of contract rules, and γ is a unique identifier of the instance.

We then define the system *configuration* as:

$$\begin{aligned}\Gamma \;= {} & \langle \mathbf{R}_1, \sigma_1, w_1 \rangle_{\gamma_1} \mid \cdots \mid \langle \mathbf{R}_n, \sigma_n, w_n \rangle_{\gamma_n} \mid \\ & \langle \mathsf{A}_1, w'_1 \rangle_{x_1} \mid \cdots \mid \langle \mathsf{A}_k, w'_k \rangle_{x_k} \mid t \end{aligned} \tag{6.5}$$

where $\langle \mathbf{R}_i, \sigma_i, w_i \rangle_{\gamma_i}$ are the contract instances, $\langle \mathsf{A}_i, w'_i \rangle_{x_i}$ are the deposits, and t is the current time. In the initial configuration there are no contract instances, and the timestamp is zero.

Configurations, as defined in (6.5), advance through *actions*. An action can perform one of the following:

- deploy a new contract instance, spending some value $w + \mathit{fee}$ from users' deposits to add $\langle \mathrm{R}, \sigma, w \rangle_\gamma$ to the configuration, where γ is a fresh identifier;
- fire a contract rule, affecting the state and balance of a contract instance, while possibly consuming and/or creating deposits;
- allow users to redistribute the tokens in their deposits;
- make the time pass.

We detail below the actions of the second kind, called contract actions. Contract actions must specify all the data needed to uniquely determine its behavior. In particular, each contract action must specify:

- the unique identifier γ of the affected instance;
- the name of the rule in $\mathbf{R}$ that is being invoked, as well as its actual parameters;
- the signers who authorize the action. There can be any number of signers (possibly zero);
- the names of each deposit that is spent to satisfy each `receive` precondition in the invoked rule;
- an additional deposit name, specifying a fee paid to fire the action. Including this fee also ensures that actions can not be replayed;
- the validity interval, constraining the time at which the rule can be performed.

We remark a few differences between the data included in our actions and those occurring in Ethereum transactions. First, our actions allow multiple signers, while in Ethereum each transaction is signed by a single externally-owned account. Second, our actions can spend multiple deposits and transfer tokens from different users to the contract. In Ethereum, the tokens transferred to the contract can only come from the sender's account. Third, our actions include an explicit validity interval, while Ethereum transactions do not. This is because Ethereum contracts can access the current block number, and then use this information to implement the wanted time constraints. The above-mentioned differences between our model and Ethereum are due to our underlying compilation target, which is a hUTXO-based blockchain instead of an account-based one.

When an action invokes a rule, if its preconditions are satisfied, the configuration is updated accordingly. The deposits spent by the action are removed from the configuration, while new deposits are added to implement the `send` effects. The contract balance w is updated based on the `receive` preconditions and `send` effects. The contract state σ is updated according to the assignments to variables and maps specified in the effect.

6.4.1 The crowdfund contract in hURF

We return to our running example to illustrate hURF: the implementation of our crowdfund contract is shown in Listing 6.1. The contract starts by listing all its state variables and maps. Map `m` tracks the amount of tokens `m[a]` that have been donated by each donor `a`. Variable `owner` stores the public key of the address that will receive the funds if the `goal` is reached. Variable `t_wd` is the deadline after which the owner can withdraw the funds, while `t_rf` is the deadline after which the donors can take their donations back, if not already claimed.

The *rules* `donate`, `withdraw`, and `refund` specify which operations can be performed on the contract. Rule `donate` can be invoked by choosing its two parameters `x` and `a`, representing the donated amount and the donor address, respectively. The `receive(x:T)` is a rule *precondition*, checking that `x` tokens have been indeed transferred to the contract upon invoking the rule. The command `m[a] = m[a]+x` is the rule *effect*: it updates the contract state, tracking the donated amount in the map `m` (note that in the initial state hURF maps are implicitly initialized so that `m[a]` is zero for all `a`).

Rule `withdraw` takes as a parameter the amount of tokens `x` that the owner

wants to withdraw from the contract. The rule precondition `require(...)` ensures that the owner has authorized this operation with their signature, that we are within the right time window, and that the owner is withdrawing enough tokens (so the goal has been reached). Finally, the rule effect `owner.send(x:T)` transfers the tokens to the owner.

The last rule `refund`s a donation if not already claimed.

We remark that hURF effects are executed *simultaneously*, not in sequence: this is why we separate them using the symbol `|` instead of a semicolon. For example, consider rule `refund`: in the effect `a.send(m[a]:T)` the value of `m[a]` is the one relative to the contract state in which the rule was invoked, and not the one affected by the other effect `m[a] = 0`. Note that executing `m[a] = 0` and `a.send(m[a]:T)` in sequence would cause zero tokens to be sent.

```
contract Crowdfund {
  map m; // donors map (from address to int)
  var owner = "pub_key_owner";
  var goal = 1000;
  var t_wd = 10; // withdraw timeout
  var t_rf = 20; // refund timeout

  donate(x,a) { // user a donates x tokens
    receive(x:T); // x:T are sent to Crowdfund
    m[a]=m[a]+x;  // donors map updated
  }
  withdraw(x) { // owner gets the entire balance
                // if the goal is reached
    require(signedBy(owner)
      && validFrom>=t_wd
      && validTo<t_rf
      && x>=goal);   // goal reached
    owner.send(x:T); // transfer x:T to owner
  }
  refund(a) { // refunds to a its donation
              // if not already claimed
    require(validFrom>=t_rf); // deadline passed
    m[a]=0 |    // resets m[a] and (in parallel)
    a.send(m[a]:T); // transfer m[a] tokens to a
  }
}
```

Listing 6.1: hURF specification of the crowdfund contract.

6.5 Compiling hURF to hUTXO

Our compiler translates hURF contracts into a set of hUTXO outputs, split in *logic outputs* and *state outputs*. For each hURF rule, it generates a logic output, whose script ensures that the rule precondition holds and that the effect is correctly applied. To this purpose, the script checks that any transaction T having the logic output in its inputs has the form in Figure 6.2, and satisfies the following conditions:

- T's inputs include all the state outputs related to the parts of the state read or written by the hURF rule. This allows the script to evaluate all the hURF expressions occurring in the rule, hence to check the `require` preconditions.
- T includes inputs providing the funds to satisfy the `receive` preconditions.
- In order to perform the effect, T spends the state outputs representing the state values which are being overwritten, while generating (in the outputs of T) new state outputs for the new state values. As discussed before, this might involve splitting and/or merging open intervals as needed.
- T's outputs include the "standard deposits" needed to transfer tokens to participants as specified by `send` effects.
- T includes an additional input for the transactions fees; further inputs and outputs in T are forbidden.

Finally, the compiler generates the state outputs for the initial state, following our distributed approach. Their script mandates that any transaction T having the state output among its inputs must also include one logic output in its inputs.

Indeed, if any transaction T attempts to update the state by creating a contract state output or to exchange tokens with the contract, the hUTXO validity conditions force T either to use a distinct contract ID (in which case, it does not affect the current contract), or to include at least one input related to the current contract ID. Such an input can only refer to state outputs or logic outputs. The script for state outputs requires the presence of an input referring to a logic output, so in all cases, a logic output must be present.

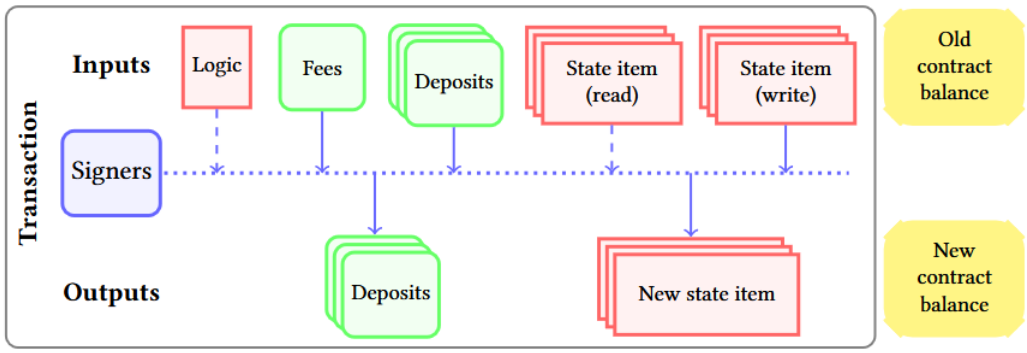


Figure 6.2: A transaction invoking a hURF rule. The inputs and outputs marked with the contract ID are depicted as red squares. Dashed lines represent non-consumed inputs. The contract balance is implicitly updated according to the difference in value between the consumed inputs and the outputs (after fees).

The script of logic outputs forces the hURF semantics to be respected, and also forbids the creation of *new* logic outputs, effectively preventing the creation of new contract rules. Since the logic output script prevents T from consuming its output, the set of logic outputs for a contract remains unchanged indefinitely. Overall, this forces any valid transaction T affecting the contract to have the structure shown in Figure 6.2.

In the rest of this section we will present more precisely how to compile hURF contracts into the hUTXO model. The compiler will have to take into account the concrete structure of the hUTXO transactions, their validity conditions, and the distribution of the contract state.

Recall that a contract with rules $\mathbf{R}$ can be instantiated multiple times. Each instance $\langle \mathbf{R}, \sigma, w \rangle_\gamma$ is associated to a unique contract identifier γ. At the hUTXO level, the rules $\mathbf{R}$ and the current state σ of the contract γ are represented by those UTXOs whose *inContract* flag is true and whose enclosing transaction has $ctrId = \gamma$. We will refer to such outputs as the *contract outputs* for γ.

We first show how to represent the contract state and balance on the ledger and how to update them. Then, we show how to create the output scripts associated to each contract rule.

Representing the contract state and balance We represent the state σ and balance w of the contract γ as follows:

- The balance w is simply stored as a Wallet within the *accounts* map of the Ledger. Formally, we have

$$accounts(\gamma) = w$$

- The state σ is instead stored across multiple unspent contract outputs for γ. To achieve this, we proceed in two steps: first, we *flatten* σ into a single map Σ from hashes to values. Then, we *distribute* Σ across multiple contract outputs.

Flattening σ to Σ The state σ associates variables `x` to values $\sigma(\texttt{x}) = v$, and maps `m` to functions $\sigma(\texttt{m}) = [(l_1, \cdots, l_n) \mapsto v] \cdots$. We flatten this structure, creating a new state map Σ that associates the hash of each variable name `x` and the hash of each map location $(\texttt{m}, (l_1, \cdots, l_n))$ to their respective values. More formally, we represent each variable association

$$\sigma(\texttt{x}) = v$$

with the association

$$\Sigma(\texttt{hash("var\_x")}) = v \tag{6.6}$$

Similarly, we represent each map association

$$\sigma(\texttt{m}) = [(l_1^1, \cdots, l_n^1) \mapsto v_1] \cdots [(l_1^k, \cdots, l_n^k) \mapsto v_k]$$

with the associations

$$\begin{aligned} &\Sigma(\texttt{hash("map\_m["@toStr(}l_1^1 \cdots l_n^1\texttt{)@"]")}) = v_1 \\ &\qquad\cdots \qquad\qquad \cdots \qquad\qquad \cdots \\ &\Sigma(\texttt{hash("map\_m["@toStr(}l_1^k \cdots l_n^k\texttt{)@"]")}) = v_k \end{aligned} \tag{6.7}$$

We let the map Σ associate all the other hashes to the *default value* 0. This representation Σ of the state σ is well defined, provided that there are no hash collisions. Since such collisions are extremely unlikely we simply assume that they will never happen. We assume that the codomain of the hash function `hash` is the open interval $(\min_H, \max_H)$. We will denote such values in hexadecimal format: $\min_H = \texttt{00}\ldots$ and $\max_H = \texttt{ff}\ldots$.

We provide an example in Table 6.1, showing how a state σ is flattened to Σ.

Encoding Σ as outputs To represent the map Σ as contract outputs we first notice that Σ takes the default value 0 everywhere, except for some hashes which we denote by $h_1 \cdots h_n$. For simplicity, let those n hashes be ordered, so that $h_1 < h_2 < \cdots < h_n$. We can then visualize Σ as follows:

$$\Sigma(h) = \begin{cases} 0 & h \in (\min_H, h_1) \\ v_1 & h = h_1 \\ \cdots & \\ 0 & h \in (h_{i-1}, h_i) \\ v_i & h = h_i \\ 0 & h \in (h_i, h_{i+1}) \\ \cdots & \\ v_n & h = h_n \\ 0 & h \in (h_n, \max_H) \end{cases} \tag{6.8}$$

We distribute Σ on the ledger by creating an output for each of the cases shown above, amounting to $2n+1$ outputs. This is exemplified in Figure 6.3.

Among the generated outputs, n are used to certify that $\Sigma(h_i)$ has a specific non-default value v_i. The remaining $n+1$ outputs are used to certify that for any h in the open interval (h_i, h_{i+1}), $\Sigma(h)$ takes the default value 0.

We define *state outputs* as follows:

- o_h is the output certifying that Σ takes a non-default value v in point h. We set the *datum* field of o_h to $[\texttt{"non-default"}, h, v]$.
- $\mathsf{o}_{h',h''}$ is the output certifying that Σ takes the default value in the open interval (h', h''). We set the *datum* field of $\mathsf{o}_{h',h''}$ to $[\texttt{"default"}, h', h'']$.

In both cases, we make these outputs have $value = 0$ and $inContract = true$. Finally, we let $script = \mathsf{script}_{state}$ as described in Section 6.5.

Table 6.1 gives an example of the process used to associate the original contract state σ to the new Σ map and then to the list of outputs that encode the state. Figure 6.3 depicts the input space of Σ and how outputs are related to that.

Reading and updating the state To implement the contract behavior, we will use suitable transaction scripts to verify that the contract evolves

Name	Value
x	3
y	0
m	$[1 \to 10][6 \to 100]$

1.) State σ of a contract.

Name	Key	Key hash h	$\Sigma(h)$
x	var_x	ed5ba7...	3
y	var_y	bd1138...	0
m[1]	map_m[1]	ea00c6...	10
m[6]	map_m[6]	9c90ac...	100

2.) Hash keys and mapped values, defining Σ.

	Default		Non-default	
Key	from hash	to hash	Key hash	value
—	0	9c90a...		
map_m[6]			9c90a...	100
—	9c90a...	ea00c...		
map_m[1]			ea00c...	10
—	ea00c...	ed5ba...		
var_x			ed5ba...	3
—	ed5ba...	fffff...		

3.) Outputs representing the contract. Since y takes the value 0, it is already represented in the default ranges.

Table 6.1: From contract state to the outputs.

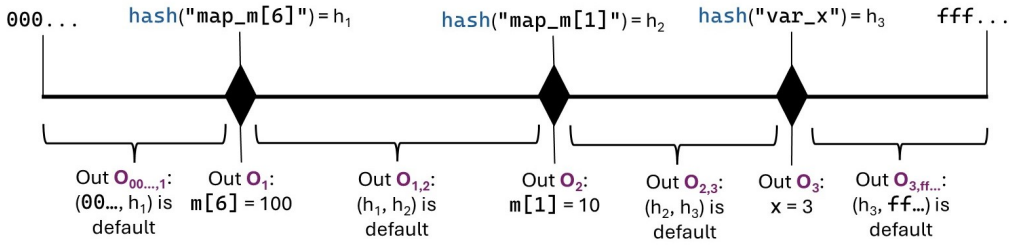


Figure 6.3: Visualizing Σ.

according to the contract rules. In order to do so, such scripts need to read and update the contract state. This is realized by making the redeeming transaction include inputs referring to those state outputs that represent the part of the state that is being accessed. More in detail, reading from the state is done by inputs having *spent* = *false*, while updating the state is done by inputs having *spent* = *true*. In the case of updates, the redeeming transaction must also include new state outputs to represent the updated state.

Reading the state In order to read the value of a contract variable `x` or of a map access `m[v,...]`, we first compute the hash h of the relevant key as in Equations (6.6) and (6.7), and then craft a transaction including among its inputs the state output that certifies the value of $\Sigma(h)$, i.e. either o_h or some o_{h_0,h_1} with $h \in (h_0, h_1)$. This allows the scripts to access the *datum* field, and read the value of $\Sigma(h)$. Note that this reading operation must not spend the output, so we set *spent* = *false* in the input.

Updating Σ: intuition To explain how the state can be updated, we first consider the case where the update only involves a single point $\Sigma(h)$, and then generalize this to the general case where multiple points are updated.

In order to update Σ on a point a transaction needs to spend the old state outputs and to create new ones. This is handled in four possible ways, depending on whether the old value $\Sigma(h) = v$ is the default or not, and on whether the new value $\Sigma'(h) = v'$ is the default or not:

- If $\Sigma(h) = v \neq 0$ and we want to update it to $\Sigma'(h) = v' \neq 0$, then the transaction needs to consume the output o_h and create a new output o'_h containing the new value v'.
- If $\Sigma(h) = v \neq 0$ and we want to update it to $\Sigma'(h) = 0$, then the transaction needs to consume three outputs $\mathsf{o}_{h_0,h}$, o_h, and o_{h,h_1} and merge them into a single range of default values, creating an output o'_{h_0,h_1}.
- If $\Sigma(h) = 0$ and we want to update it to $\Sigma'(h) = v' \neq 0$, then the transaction needs to consume the single output o_{h_0,h_1} (where $h_0 < h < h_1$), and produce three outputs: $\mathsf{o}'_{h_0,h}$, o'_h, and o'_{h,h_1}, splitting the default range into two and adding a new non-default output o'_h containing the new value v'.

- Finally, if $\Sigma(h) = 0$ and we want to update it to $\Sigma'(h) = 0$ (this can happen due to the effect of a rule), the transaction does not need to consume nor create any output. However a non-spending input is still needed to ensure that the old value is actually the default: this input is o_{h_0,h_1}, where $h_0 < h < h_1$.

Problems with updating the state on multiple points If we need to update Σ on multiple points, we might be tempted to handle each update independently, applying the technique presented above on each point. However, this does not always work.

For instance, let Σ be such that $\Sigma(h') = 1$ and $\Sigma(h'') = 2$, with $h' < h''$ and $\Sigma(h) = 0$ for any other h. If we try to update both h' and h'' to 0 within a single transaction, we would need to redeem output $\mathsf{o}_{h',h''}$ twice (once to update $\Sigma(h')$ and once again to update $\Sigma(h'')$). This is already problematic. Furthermore, the first update would create the output $\mathsf{o}_{\min_H,h''}$ while the second one would create the output $\mathsf{o}_{h',\max_H}$. However, these intervals overlap, so the new outputs would fail to correctly represent the updated state $\Sigma(')$. Indeed, a correct transaction should instead produce a single output $\mathsf{o}_{\min_H,\max_H}$.

This is not the only case in which these simultaneous updates are problematic: consider the opposite case, in which Σ is 0 everywhere, and we want a single transaction to change both $\Sigma(h')$ and $\Sigma(h'')$ to 1. Following the technique described above, this transaction would produce both the triple of state outputs $\mathsf{o}_{\min_H,h'}$, $\mathsf{o}_{h'}$, $\mathsf{o}_{h',\max_H}$, and another triple $\mathsf{o}_{\min_H,h''}$, $\mathsf{o}_{h''}$, $\mathsf{o}_{h'',\max_H}$. Once again, this does not correctly represent the updated state. Indeed, the output $\mathsf{o}_{\min_H,h''}$ would wrongly claim that $\Sigma'(h') = 0$ (and similarly $\mathsf{o}_{h',\max_H}$ would claim $\Sigma'(h'') = 0$).

Updating Σ in the correct way Here we detail an algorithm that, given a list of updates $\boldsymbol{u} = (h_1 \mapsto v_1) \cdots (h_n \mapsto v_n)$ and an initial state Σ, constructs a list of inputs $\boldsymbol{in}$ and a list of outputs $\boldsymbol{out}$ that a transaction would need to include in order to perform the updates in $\boldsymbol{u}$ to the state. Intuitively, this algorithm suitably splits and/or joins the intervals affected by the updates.

We assume that the list of updates $\boldsymbol{u} = (h_1 \mapsto v_1) \cdots (h_n \mapsto v_n)$ given as input to our algorithm is sorted, so that if $i < j$ then $h_i < h_j$. Note that, by requiring a strict inequality, we effectively forbid multiple updates at the same point (consistently with the fact that we want to perform all of them

simultaneously).

We start by constructing the list of inputs $\boldsymbol{in}$ affected by the updates $\boldsymbol{u}$. Given an initial state Σ, the algorithm repeats the following steps to add inputs to the initially empty list $\boldsymbol{in}$.

1. If $\boldsymbol{u}$ is empty, terminate.
2. If $\boldsymbol{u} = (h \mapsto v)\boldsymbol{u'}$, with $\Sigma(h) \neq 0$ and $v \neq 0$, then append o_h to $\boldsymbol{in}$ and remove $(h \mapsto v)$ from $\boldsymbol{u}$.
3. If $\boldsymbol{u} = (h \mapsto 0)\boldsymbol{u'}$, with $\Sigma(h) \neq 0$, then find h', h'' for which there exist outputs $\mathsf{o}_{h',h}, \mathsf{o}_h, \mathsf{o}_{h,h''}$. Append $\mathsf{o}_{h',h}$ to $\boldsymbol{in}$ unless it already occurs in it. Then append o_h, and $\mathsf{o}_{h,h''}$ to $\boldsymbol{in}$. Finally, remove $(h \mapsto 0)$ from $\boldsymbol{u}$.
4. If $\boldsymbol{u} = (h \mapsto v)\boldsymbol{u'}$, with $\Sigma(h) = 0$ then find h', h'' for which there exists an output $\mathsf{o}_{h',h''}$ with $h' < h < h''$. Append $\mathsf{o}_{h,h'}$ to $\boldsymbol{in}$ unless it already occurs in it. Then remove $(h \mapsto v)$ from $\boldsymbol{u}$.

Constructing the outputs list $\boldsymbol{out}$ is done by the partial function $\mathfrak{Out}$ defined in Figure 6.6. When called as $\mathfrak{Out}(\boldsymbol{in}, \boldsymbol{u})$ the function checks whether the list of inputs $\boldsymbol{in}$ can support the state updates $\boldsymbol{u}$, and in such case it returns the list of outputs $\boldsymbol{out}$ to be included in the transaction to drive the wanted state update. Intuitively, this function iterates in parallel over the lists $\boldsymbol{in}$ and $\boldsymbol{u}$, finding the points where the input intervals need to be split or merged, producing suitable outputs in the process.

For instance, let $h_1 < h_2 < h_3 < h_4$, and consider the following state Σ:

$$\Sigma(h) = \begin{cases} 1 & \text{if } h = h_1 \\ 3 & \text{if } h = h_3 \\ 4 & \text{if } h = h_4 \\ 0 & \text{otherwise} \end{cases}$$

Let $\boldsymbol{u} = (h_2 \mapsto 2)(h_3 \mapsto 0)$ be the wanted state updates. Assume that Σ is currently represented by the UTXOs $\mathsf{o}_{\min_H,h_1}$, o_{h_1}, o_{h_1,h_3}, o_{h_3}, o_{h_3,h_4}, o_{h_4}, $\mathsf{o}_{h_4,\max_H}$. We generate a transaction to update the state. Running the input generation algorithm we obtain the list of inputs $\boldsymbol{in} = \mathsf{o}_{h_1,h_3}\mathsf{o}_{h_3}\mathsf{o}_{h_3,h_4}$. Running the output generation algorithm as $\mathfrak{Out}(\boldsymbol{in}, \boldsymbol{u})$ we obtain $\boldsymbol{out} = \mathsf{o}'_{h_1,h_2}\mathsf{o}'_{h_2}\mathsf{o}'_{h_2,h_4}$. Indeed, the algorithm splits o_{h_1,h_3} to account for h_2 being changed from the default value to 2, and merges o_{h_3} into the output o'_{h_2,h_4} representing a new default interval.

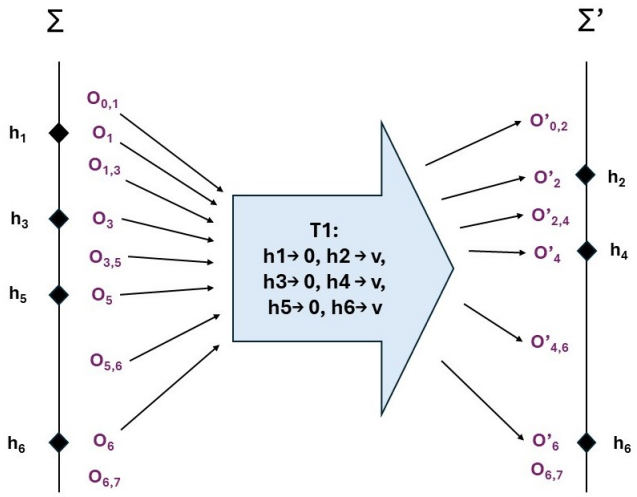


Figure 6.4: The state can be modified drastically with a single transaction.

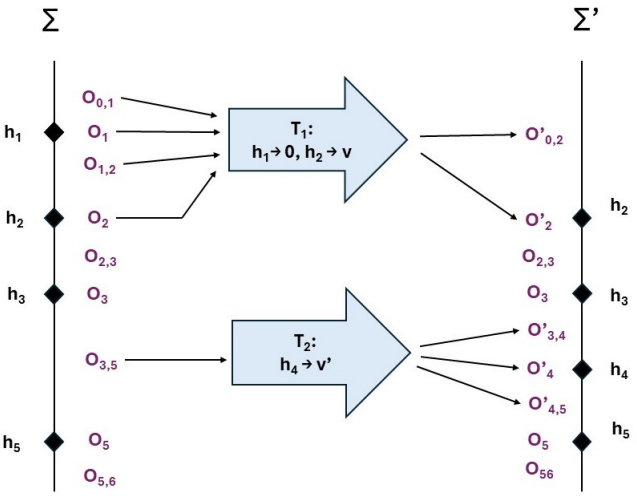


Figure 6.5: As long as they do not consume the same part of the state, transactions can be parallelized.

Example: multiple state updates We show in Figure 6.4 the effect of a transaction T_1 that updates the state at six distinct points, consuming eight state outputs while spawning six new ones. The inputs and outputs in the figure are those computed by the input and output generation algorithms for the updates

$$\boldsymbol{u} = (h_1 \mapsto 0, h_2 \mapsto v, h_3 \mapsto 0, h_4 \mapsto v, h_5 \mapsto 0, h_6 \mapsto v)$$

with $v \neq 0$. Taken individually, the update part $(h_1 \mapsto 0)$ would require the state outputs $\mathsf{o}_{0,1}, \mathsf{o}_1, \mathsf{o}_{1,3}$ to be merged, but the update part $(h_2 \mapsto v)$ would instead require $\mathsf{o}_{1,3}$ to be split. The net effect of $(h_1 \mapsto 0, h_2 \mapsto v)$ is that the new outputs include $\mathsf{o}'_{0,2}, \mathsf{o}'_2$. A similar interaction is caused by $(h_3 \mapsto 0, h_4 \mapsto v)$. By contrast, the update part $(h_5 \mapsto 0, h_6 \mapsto v)$ does not require a split, since o_6 already associates h_6 to a non-default value.

We remark that the updates $\boldsymbol{u}$ modify the state simultaneously, therefore we bundle them in a single transaction. Indeed, regardless of the size of $\boldsymbol{u}$, the inputs and outputs generation algorithms always craft a single transaction that performs them all.

Example: parallel updates Figure 6.5 shows how two transactions updating the state at different points can be validated in parallel. There, T_1 updates the state at points h_1, h_2 while T_2 updates the state at point h_4. The inputs of the two transactions do not overlap, so a parallel validation is possible.

Note that updating the state at different points is not, in general, sufficient to guarantee that two transactions are parallelizable. Indeed, had T_1 performed the update $(h_3 \mapsto 0)$, we would still have disjoint update points (h_3 vs. h_4) but both transactions would attempt to consume $\mathsf{o}_{3,5}$ (T_1 to merge it with $\mathsf{o}_{2,3}$, T_2 to split it). This makes the two transaction conflict: only one of them can be appended to the current blockchain, and their validation can not be made in parallel. After one of them is put on the blockchain, the other update can still be performed, but using a different transaction involving the new UTXOs.

Contract logic We now show how to implement the contracts of Section 6.4 on the blockchain. In Section 6.5 we saw how to represent, read and update the state. Here, starting from a set of rules $\mathbf{R}$, we create a set of contract outputs that constrain the state updates to follow the rules.

$$\mathfrak{Out}^*(\varepsilon, \varepsilon) = \varepsilon$$
$$\mathfrak{Out}^*(\mathsf{o}_{h',h''}, \varepsilon) = \mathsf{o}_{h',h''}$$

$$\mathfrak{Out}^*(\mathsf{o}_{h',h''}\boldsymbol{in}, (h \mapsto v)\boldsymbol{u}) =$$
$$= \begin{cases} \mathsf{o}_{h',h''}\mathfrak{Out}(\boldsymbol{in}, (h \mapsto v)\boldsymbol{u}) & \text{if } h > h'' \\ \mathfrak{Out}(\mathsf{o}_{h',h''}\boldsymbol{in}, (h \mapsto v)\boldsymbol{u}) & \text{otherwise} \end{cases}$$

$$\mathfrak{Out}(\varepsilon, \varepsilon) = \varepsilon$$

$$\mathfrak{Out}(\mathsf{o}_{h',h''}\boldsymbol{in}, (h \mapsto v)\boldsymbol{u}) =$$
$$= \begin{cases} \mathfrak{Out}^*(\mathsf{o}_{h',h''}\boldsymbol{in}, \boldsymbol{u}) & \text{if } v = 0,\ h \in (h', h'') \\ \mathsf{o}_{h',h}\mathsf{o}_h\mathfrak{Out}^*(\mathsf{o}_{h,h''}\boldsymbol{in}, \boldsymbol{u}) & \text{if } v \neq 0,\ h \in (h', h'') \end{cases}$$

$$\mathfrak{Out}(\mathsf{o}_h\boldsymbol{in}, (h \mapsto v \neq 0)\boldsymbol{u}) = \mathsf{o}'_h\mathfrak{Out}(\boldsymbol{in}, \boldsymbol{u})$$
$$\mathfrak{Out}(\mathsf{o}_{h',h}\mathsf{o}_h\mathsf{o}_{h,h''}\boldsymbol{in}, (h \mapsto 0)\boldsymbol{u}) = \mathfrak{Out}^*(\mathsf{o}_{h',h''}\boldsymbol{in}, \boldsymbol{u})$$

Figure 6.6: Definition of the partial function $\mathfrak{Out}$. In those cases not explicitly handled by the equations above, the operation fails. The definition uses the auxiliary function $\mathfrak{Out}^*$.

To deploy a contract, we append a transaction T_{init} with a fresh contract identifier, as required by item 7 of the transaction validity conditions (see page 115). The transaction T_{init} contains a contract output o^{R} for each contract rule R, as well as a set of state outputs to represent the initial contract state, as explained in Section 6.5.

Once the contract is deployed, one can execute a rule R by appending a transaction T referring to the output o^{R} as a non-spending input. This causes the script $\mathsf{o}^{\mathrm{R}}.script$ to be executed. We define such script so to verify that T correctly updates the current contract state as mandated by rule R. This is done by inspecting all the inputs and the outputs of T. In Figure 6.2 we show the structure of T.

We now describe o^{R} in more detail. We set its fields as follows: $value = 0$, $datum =$ `"logic"`, $inContract = true$. Further, $\mathsf{o}^{\mathrm{R}}.script$ verifies that:

- o^{R} is included as the first input in the redeeming transaction. This ensures that only one rule (i.e., R) is executed by the redeeming transaction T.
- The first input of T specifies $spent = false$. Overall, this ensures that o^{R} can never be consumed.
- The second input of T spends a non-contract output and its value is equal to $\mathsf{T}.fee$. This allows the input to pay transaction fees without affecting the contract state and balance.
- If k is the cumulative number of variables and map accesses that are read by rule R, then the script checks that the next k inputs specify $spent = false$ and refer to contract outputs with a `"default"` or `"non-default"` datum, reading the values in the process. The script checks that the hash-keys in the input datums are the expected ones (so that we do not read some other part of the state). Moreover, the rule parameters are read from the *redeemer* field of the first input of T. After this step, the script is able to evaluate all the expressions occurring in R.
- If n is the number of `receive` preconditions of R, then the script checks that the next n inputs spend non-contract outputs contributing the exact values specified in those `receive` preconditions.
- The expression in the `require` precondition of R evaluates to *true*.

- If m is the number of `send` statements in R, then the script checks that the first m outputs in T set $inContract = false$ and pay the intended amount of tokens to the intended recipients, as specified by the rule.

- Finally, the script checks that the remaining inputs and outputs perform the state update mandated by the rule. To this purpose, the script first computes the list of the remaining inputs $\boldsymbol{in}$ and the list of state updates $\boldsymbol{u}$. The script checks that the inputs within $\boldsymbol{in}$ have $spent = true$. Then it computes $\boldsymbol{out} = \mathfrak{Out}(\boldsymbol{in}, \boldsymbol{u})$ as per Figure 6.6. If that fails, then T is rejected. Otherwise, the script verifies that the remaining outputs of T are equal to $\boldsymbol{out}$. In this way, this step ensures that there are no extraneous inputs or outputs w.r.t. those needed to implement the intended state update.

By construction, when the "logic" output o^{R} is referred as an input in T, the contract state must be updated according to rule R. We prevent the contract state to be modified in any other way from those specified by the rules. This is done by making the script script_{state} of all state outputs to require the presence of some "logic" output o^{R}. More precisely, script_{state} checks that in the redeeming transaction T the first input refers to a contract output whose *datum* is `"logic"`.

Example Consider a contract with variables `a,w,y,z` and map `m`, having the following rule:

```
example(x) {
  receive(z:T0);
  require(z>10);
  w=m[x]-y | m[z]=7+m[1] | a.send(1:T1);
}
```

Consider the execution of the rule in the state σ where:

$$\sigma(\mathtt{y}) = 3 \quad \sigma(\mathtt{w}) = 2 \quad \sigma(\mathtt{z}) = 15 \quad \sigma(\mathtt{a}) = \mathtt{"pubkey\_a"}$$
$$\sigma(\mathtt{m}) = [14 \mapsto 1][27 \mapsto 3]$$

and let the contract balance be $w = [100{:}\,\mathrm{T}_1]$.

The flattened state map Σ takes as input the hashes of variable names and map location. These are generated as in Equations (6.6) and (6.7). Assume

that they are ordered as follows:

$$\begin{aligned}\min_H < h_y < h_w < h_z < h_a < \\ < h_{m[1]} < h_{m[14]} < h_{m[15]} < h_{m[27]} < \max_H\end{aligned}$$

We now craft a transaction T that executes the rule with parameter $\mathtt{x} = 27$. The first input of T must refer to the output $\mathsf{o}^{\mathtt{example}}$ that encodes the rule logic. Moreover, this input must have *redeemer* set to 27, specifying the value of `x`. The second input of T must spend a previous output contributing the fees. Inputs from the third to the seventh access (without spending them) the state outputs relative to the values read by the contract. More precisely, we have: o_{h_y} for `y`, o_{h_z} for `z`, o_{h_a} for `a`, $\mathsf{o}_{h_a,h_{m[14]}}$ for `m[1]` (since $h_a < h_{m[1]} < h_{m[14]}$), and $\mathsf{o}_{h_{m[27]}}$ for `m[x]`, in that order. The values stored in the outputs referenced from these inputs are then used throughout the script to evaluate the expressions. The eight input must spend an output contributing 15: T_0 to satisfy the `receive(z:T0)` precondition. Finally, inputs from the ninth to the twelfth spend the outputs relative to the state updates. Note that the updates are $\boldsymbol{u} = (h_w \mapsto 0), (h_{m[15]} \mapsto 7)$ since in the state σ the expression `m[x]-y` evaluates to $3 - 3 = 0$, while `z` evaluates to 15 and `7+m[1]` evaluates to $7 + 0 = 7$. To achieve the wanted update $\boldsymbol{u}$, we include in T the inputs $\boldsymbol{in}$ specified by the input generation algorithm of Section 6.5. These include $\mathsf{o}_{h_y,h_w}, \mathsf{o}_{h_w}, \mathsf{o}_{h_w,h_z}$ which are needed for $h_w \mapsto 0$ and $\mathsf{o}_{h_{m[14]},h_{m[27]}}$ which is needed for $h_{m[15]} \mapsto 7$. These inputs have *spent* set to *true*.

We also generate the outputs for the updates, according to $\mathfrak{Out}(\boldsymbol{in}, \boldsymbol{u})$. Since $h_w \mapsto 0$ causes `w` to change from a non-default value to the default one, the inputs $\mathsf{o}_{h_y,h_w}, \mathsf{o}_{h_w}, \mathsf{o}_{h_w,h_z}$ are merged into a single new output o'_{h_y,h_z}. Instead, since $h_{m[15]} \mapsto 7$ causes `m[15]` to change from the default value to a non-default one, the input $\mathsf{o}_{h_{m[14]},h_{m[27]}}$ is split into three new outputs $\mathsf{o}'_{h_{m[14]},h_{m[15]}}$, $\mathsf{o}'_{h_{m[15]}}$, and $txo'_{h_{m[15]},h_{m[27]}}$.

The outputs of T also include an output to implement the `send` statement, transferring 1: T_1 to the public key `"pubkey_a"` (the result of the evaluation of `a`). This output has *inContract* set to false, and the owner of the corresponding private key can freely spend it.

After the transaction is appended to the blockchain, the state of the contract is updated to

$$\sigma'(\mathtt{y}) = 3 \quad \sigma'(\mathtt{w}) = 0 \quad \sigma'(\mathtt{z}) = 15 \quad \sigma'(\mathtt{a}) = \mathtt{"pubkey\_a"}$$
$$\sigma'(\mathtt{m}) = [14 \mapsto 1][15 \mapsto 7][27 \mapsto 3]$$

and the contract balance is updated to:

$$w' = [15\colon \mathsf{T}_0, 99\colon \mathsf{T}_1]$$

Contract deployment To deploy a new contract instance $\langle \mathbf{R}, \sigma, w \rangle_\gamma$ we need to append a suitable transaction T_{init} to the blockchain. The first input of T_{init} must spend a previous non-contract output, contributing the fees. We set $\mathsf{T}_{init}.fee$ to its value. As specified by the validity conditions (item 7 at page 115), the hash of this input also determines a fresh contract identifier γ. Accordingly, we set $\mathsf{T}_{init}.ctrId$ to this value. We then fill the outputs of T_{init}, adding one contract output o^{R} for each rule $\mathrm{R} \in \mathbf{R}$, as well as the sequence of state outputs $\mathsf{o}_{\min_H, h_0}, \mathsf{o}_{h_0}, \mathsf{o}_{h_0,h_1}, \mathsf{o}_{h_1}, \ldots, \mathsf{o}_{h_n,\max_H}$ according to the initial state σ of the contract. These outputs have zero *value*, and have as their scripts the ones described before.

If the required initial balance w is non-zero, we also include in T_{init} one or more (non-contract) spent input, whose overall value amounts to w. Since all the outputs of T_{init} have zero value, the currency received from the inputs is transferred to the contract balance. The *redeemer* fields of all the inputs is set so to satisfy the scripts of the outputs they are spending. The *spent* field of all the inputs is set to *true*. Finally, T_{init} is signed by all the users whose authorization is required to spend the inputs.

6.6 Discussion on hURF and improvements

We now discuss several aspects of our approach, as well as a few possible variants.

Security of the compiler The security of our approach hinges on precisely controlling the form of the contract outputs, i.e. those UTXOs having $inContract = true$ and the contract $ctrId = \gamma$ in their transaction. Indeed, it is crucial that such UTXOs correctly represent the current contract state as well as the logic operations corresponding to contract rules. More specifically, the contract outputs must either have $datum =$ `"logic"` and check the correct application of a contract rule, or have a *datum* of one the forms `"default"`/`"non-default"` and contribute to represent the contract state as in Equation (6.8).

Breaking this invariant would allow a malicious user to disrupt the execution of the contract. Adding a new `"logic"` contract output allows updating the contract state without following the contract rules. Removing a `"logic"` contract output would instead prevent the execution of a contract rule. Moreover, adding a spurious state output (`"default"`/`"non-default"`) could cause confusion on the value of a contract variable or map location: reading the state could then use one of the two different values on the blockchain, potentially leading to vulnerabilities. Finally, removing a state output in an uncontrolled way can prevent reading and updating that part of the state, effectively preventing the execution of the rules which attempt that.

This invariant on the contract outputs is enforced at multiple levels. First, our blockchain model prevents users from freely creating new UTXOs in an existing contract. To see why, assume we have an existing contract with $ctrId = \gamma$. The validation conditions for new transactions (see page 115) allow new contract outputs to be created only in two cases:

- Item 6 allows to create new contract outputs o', but only when at least one input refers to a previous contract output o having the same *ctrId*. Hence, one can create contract outputs o' with $ctrId = \gamma$ only when the previous script $\mathsf{o}.script$ accepts it.

- Item 7 allows to create new contract outputs o', but only when there are no input referring to a contract output o, and when the *ctrId* of o' is generated from a spent input. This makes *ctrId* a fresh contract

identifier because of the no-double-spending property. Hence, this *ctrId* must be different from γ, making the contract output o' belong to a new contract.

Consequently, the creation of new contract outputs inside $ctrId = \gamma$ can only be done with the permission of one of the previous output scripts within the contract. This allows to enforce our invariant at the script level, by making the contract-related scripts verify the newly created contract outputs. Indeed, no new `"logic"` outputs are accepted, and the new `"default"`/`"non-default"` outputs are only accepted when they are part of a state update mandated by some contract rule.

Simplicity of the scripts As discussed in Section 6.5 our scripts have to perform several checks on the fields of the redeeming transaction T and the UTXOs referred to by T. We remark that such checks do not involve any other parts of the blockchain, so they can be performed within the minimalistic script model described in Section 6.3.

We note that the mentioned checks would also be feasible in a loop-free script language. Indeed, the most complex check our scripts need to perform is the one on the outputs defined in Figure 6.6. While that algorithm is presented as a recursive function, the number of recursive calls it performs is at most $2n$ where n is the number of the assignments in the contract rule effect, which is statically known. It is therefore feasible to unfold the recursion up to that level and obtain an equivalent loop-free script.

Improving the compiler The transactions produced by our compiler can be optimized and made slightly more efficient in a few scenarios. For instance, when updating the state according to $\boldsymbol{u} = (h \mapsto v)$ it is possible that the new value v is actually the same as the old value. Our compiler always consumes the old state output o and creates a new one o', even the value does not change. In such case, it would suffice reading o using a transaction input with $spent = false$ and check that the old value is indeed v. Doing so would be more efficient since it would generate smaller transactions (one fewer output), possibly reducing the fees. Further, it would increase the opportunities of parallelization by avoiding conflicts with other transactions reading o (see Section 6.7).

Our compiler can also generate multiple inputs for the same part of the state, as it happens when a contract variable is both read and updated. This

could be optimized by omitting the redundant inputs, at the cost of some additional complexity in the scripts.

Closing contracts Our contracts do not feature any means of termination. Once deployed, a contract lives forever: its logic outputs and state outputs are never fully removed from the UTXO set. At best, the number of state outputs can decrease when setting some part of the state to the default value. We could extend our contract model and compiler to allow a complete shutdown of the contract. To achieve that, a special `"logic"` output could verify that a suitable programmed precondition for the shutdown is satisfied, and instantly consume all the `"logic"` outputs, effectively forbidding any new firing of contract rules. At the same time, a new `"logic"` output o_{del} is spawned to start the consumption of all the existing state outputs, following the order given by the hashes stored in their *datum* fields, from $\min_H$ to $\max_H$. This can be realized by making o_{del} keep the hash h of the next state output to consume in $\mathsf{o}_{del}.datum$. The only way to spend o_{del} would then be to consume the outputs $\mathsf{o}_{del}, \mathsf{o}_{h,h'}, \mathsf{o}_{h'}$ (for any h') and produce a new output o'_{del} with a new hash h' in its *datum*. In this way, the deletion process can continue until $\max_H$ is found, at which point no new output needs to be produced, completing the contract shutdown.

Avoiding spam transactions Our logic and state outputs have their *value* field set to 0, so that creating a new state output only costs the transaction *fee*. This might allow contracts to generate an excessively large amount of UTXOs, polluting the UTXO set that validator nodes must handle. To incentivize a more conservative use of state storage, fees could be increased for state outputs. For instance, the fee could be made proportional to the transaction size as in Cardano [64].

Alternatively, a small "collateral" value could be deposited in the state outputs' *value*. Such collateral could then be refunded when the state output is consumed (including when the contract is shut down). This would be analogous to the "rent" mechanism used by Solana [119].

6.7 Parallel validation of transactions

6.7.1 Conflicts

In order to parallelize the validation of transactions, it is paramount to be able to understand the nature of conflicts that can hinder or prevent parallel execution. Such conflicts can be observed at the contract (hURF) level or at the transaction (hUTXO) level.

A first kind of conflict is what we call a *hard conflict*, and is caused by two contract operations that can not be performed both, either in sequence or in parallel. A common case of hard conflict happens when one operation alters the contract state making it violate the precondition of the other operation, and vice versa. Another case of hard conflict is due to operations wanting to spend tokens from the contract balance, which is sufficient for either operation but not both.

By contrast, a *soft conflict* does allow two operations to be run both, just not in parallel. A simple soft conflict is caused by one operation writing a part of the contract state which has to be read by the other operation. This kind of data dependency forces the operations to be run in sequence.

Note that, sometimes, at the transaction level we can observe more conflicts that do not show up at the contract level. For instance, consider a contract with rule `f(v) { x = v; }`, which is being invoked as `f(1)` and `f(2)`. At the transaction level, both calls would attempt to spend the same state output, so they cause a hard conflict.

As another example, consider rules `set_x(v) { x = v; }` and `set_y(v) { y = v; }`. If current contract state associates to each variable the default value, then attempting to call `set_x(1)` and `set_y(2)` causes a hard conflict, even if they affect disjoint variables. This is because both variables are represented by the *same* state output $o_{\min_H,\max_H}$, hence a double spending hard conflict appears.

6.7.2 Parallel validation

Our validator takes as input a sequence of transactions, constructs its longest conflict-free prefix, and sends it to a pool of worker threads for validation. Then, it commits to the blockchain the transactions that passed validation, so updating the blockchain state, and restarts by processing the next prefix.

Detecting conflicts in our hUTXO model can be done efficiently. Note that we can focus on transaction-level conflicts, since each contract-level

conflict is also a conflict at the level of transactions. Finding double spending attempts is straightforward since each transaction explicitly mentions the outputs it is trying to consume or to read (through inputs with *spent* = *true* or *false*, respectively).

Dependency conflicts (either hard or soft) can also be detected in a similar fashion, by examining the explicit inputs and outputs of transactions.

For balance-related conflicts, we adopt a conservative execution approach. We first compute the tokens that each transaction is going to consume from the contract balance, as specified in Equation (6.3). Then, if such tokens are still available in the contract balance, we make a thread validate the transaction while temporarily marking those tokens as frozen, making them unavailable for other transactions. If the transaction is ultimately accepted, the tokens are consumed, while on rejection they are thawed. We call our approach as conservative since we only validate transactions when we are sure there is enough contract balance for them. This approach is similar to the banker's algorithm for resource allocation, with the key difference that in this case allocated resources (tokens) are not necessarily freed after they have been used: once a transaction consumes the tokens, they are completely gone from the contract balance.

Our parallel validation algorithm is efficient. Beyond the standard execution of scripts and verification of digital signatures, it requires only a modest amount of additional bookkeeping for the affected UTXOs and the contract balances. If we use a balanced search tree to store those, we only pay an additional logarithmic cost for each input and output, and each involved contract. This can be further improved using hash tables, obtaining a constant overhead.

6.8 Experimental evaluation

To assess our approach, we have implemented a prototype tool to measure the performance of hUTXO validation, and applied it to a benchmark of use cases. More specifically, our tool, named DiSCo_sim, features:

- a blockchain node simulator for our hUTXO model, comprising two alternative transaction validators: a *sequential* one that only uses one thread, and a *parallel* one that can exploit a thread pool;
- routines to generate the transactions to deploy and execute hURF contracts, coherently with our compiler;
- a benchmark of use cases, discussed in detail below, comprising the crowdfund contract introduced in Section 6.4.1, a minimalistic map contract, a multisig wallet, and a registry contract.

To make our experimental evaluation realistic, we use the same cryptographic primitives as Cardano, i.e. BLAKE2b-512 for hashing, and Ed25519 for signatures. Since we do not aim at developing a full blockchain node, but only a proof-of-concept to validate our hUTXO model, we decided to simplify some aspects of our simulator, so to keep the required development effort contained. In particular, this implies that our simulator diverges from Cardano in some aspects.

First, DiSCo_sim only focusses on the execution of contracts, and does not include those functionalities of validators that are orthogonal to that aim (e.g., the consensus protocol).

Second, we simplify the generation of transactions in our experiments in two aspects:

- We use sequential numbers instead of hashes as transaction IDs, since this simplifies the generation of chains of transactions that refer to each other. The actual cryptographic hash function is still used for all other purposes, including reading and updating the contract state as detailed in Section 6.5, and in particular in Equations (6.6) and (6.7) which define the expected hashes that occur in state outputs.
- We do not actually sign the transactions. Instead, whenever a signature on a transaction would need to be checked, DiSCo_sim verifies a fixed signature on a fixed message. We ensure that this operation

is not optimized away during compilation, so as not to invalidate the performance measurements.

We believe that, even with these simplifications, the performance measures on our simulator should be close to a full implementation of a hUTXO validator node, and therefore the results of our experiments would still give a reasonable estimate of the actual performance in the wild.

Our simulator was developed in Rust, and amounts to ∼3500 Lines of Code and 125KB. The experiments code instead amounts to 2200 LoC and 73KB.

6.8.1 Experimental setup

For each benchmark we setup and execute a series of experiments, each comprising the following actions:

1. We fix the number $N_{threads}$ of worker threads available for the node, beyond a main thread.

2. We generate a sequence $\boldsymbol{T}$ of transactions to be validated, representing the interactions of the users with the contracts considered in the experiment. This sequence does not include *hard conflicts*, but does include some *soft conflicts* in the form of transaction dependencies, i.e. transactions whose inputs refer to the outputs of previous transactions in the sequence. This choice is coherent with simulating a *validator* that is verifying proposed blocks whose transactions have a very high chance to be free from hard conflicts. By contrast, we do not simulate *miners* that build blocks selecting transaction from the mempool, which instead contains arbitrary transactions from the users, often involving hard conflicts.

3. We start a timer, and proceed to validate the transactions in the sequence $\boldsymbol{T}$. More precisely:

   - If $N_{threads} = 0$ we run the *sequential* validator, which processes the transactions in $\boldsymbol{T}$ one by one.
   - Otherwise, a pool of $N_{threads}$ worker threads is created. The main thread repeatedly scans the sequence $\boldsymbol{T}$, searching for the longest prefix of conflict-free transactions (as detailed in Section 6.7), and

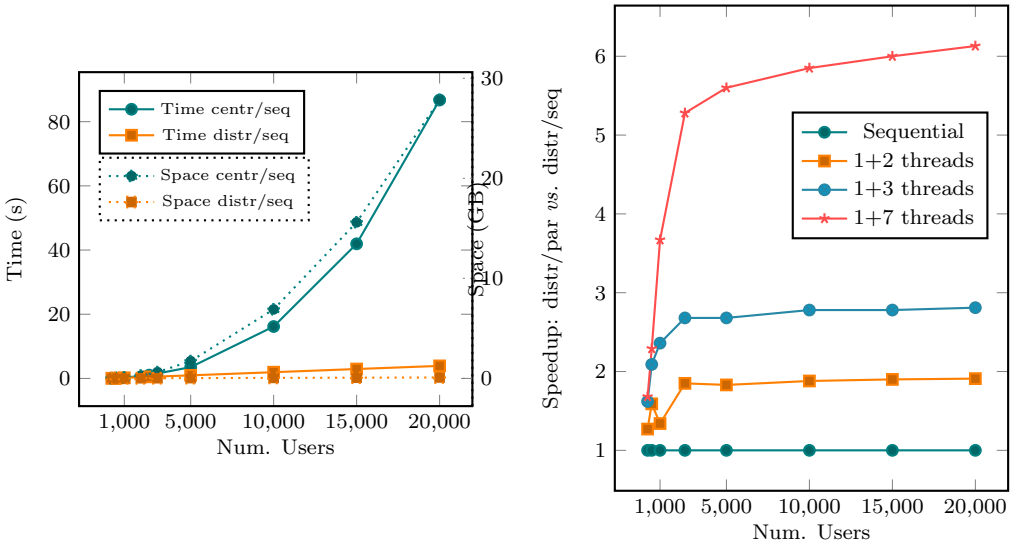


Figure 6.7: Time, space, and speedup for the Crowdfund experiments.

sends them to the worker threads for validation. When the worker threads complete a batch of transactions, the main thread commits the valid transactions in the batch on the ledger.

4. Finally, after the whole sequence $\boldsymbol{T}$ has been processed, we stop the timer and report its value.

All the measures are the average over 10 runs on a 3GHz 64-bit Intel Xeon Gold 6136 CPU and a GNU/Linux OS (x86_64-linux) with 8 cores and 64 GB of RAM.

6.8.2 Results

The common goal of our experiments is to measure the speedup obtained by distributing the contract state and by parallelizing the validation of transactions. To this purpose, we consider a benchmark of contracts with different features. For each use case, we design a set of experiments varying the number of available threads as well as other parameters specific to the use case (e.g., number of users, percentage of conflicts, etc.). We then measure both time and space usage for each experiment. For the sake of clarity, we illustrate a representative subset of the experimental data in the plots below; the full data is available at [29].

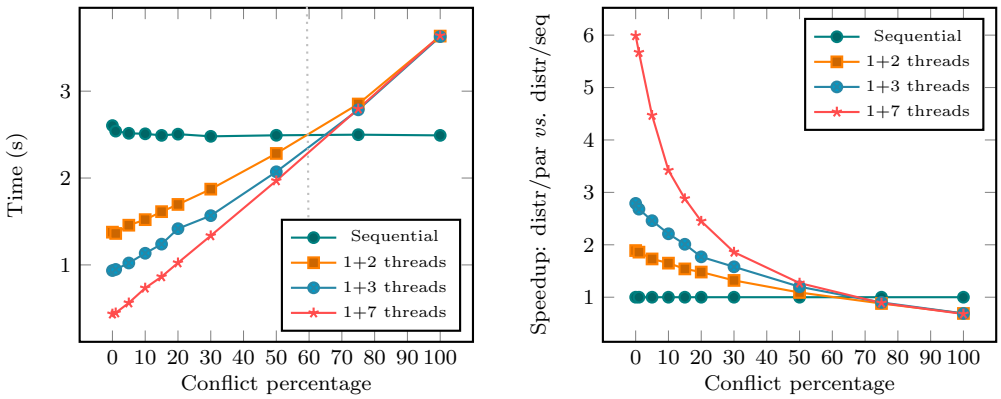


Figure 6.8: Timings and speedup for the Map experiments.

Validator	C(250)	D(250)	C(500)	D(500)	C(1K)	D(1K)
Sequential	0.040	0.047	0.092	0.094	0.244	0.191
1+1 threads	0.057	0.057	0.110	0.102	0.268	0.201
1+2 threads	0.067	0.037	0.117	0.059	0.280	0.143
1+3 threads	0.078	0.029	0.116	0.045	0.281	0.081
1+4 threads	0.087	0.026	0.147	0.038	0.281	0.064
1+5 threads	0.069	0.024	0.118	0.038	0.311	0.056
1+6 threads	0.057	0.026	0.129	0.041	0.321	0.052
1+7 threads	0.056	0.028	0.138	0.041	0.320	0.052

Validator	C(10K)	D(10K)	C(20K)	D(20K)	D(50K)	D(1M)
Sequential	16.172	1.943	86.758	3.908	10.827	200.446
1+1 threads	11.283	2.011	41.257	4.061	10.008	200.656
1+2 threads	11.702	1.033	45.100	2.051	5.088	101.515
1+3 threads	13.653	0.698	67.941	1.389	3.441	67.976
1+4 threads	13.236	0.539	61.785	1.063	2.609	51.101
1+5 threads	14.744	0.461	77.655	0.863	2.104	41.046
1+6 threads	15.176	0.375	77.371	0.741	1.786	34.335
1+7 threads	17.385	0.332	81.322	0.638	1.544	29.528

Table 6.2: Timings for the Crowdfund experiments (C=centralized, D=distributed).

Crowdfund We experiment with the crowdfund contract of Section 6.2.1. The interest in this contract is that it includes methods that simultaneously update maps and transfer tokens, but still in a way that leads to an efficient parallelization. This was expected, because donors operate on disjoint parts of the state, and token transfers from the contract are always successful, so reducing conflicts.

We craft experiments for the centralized and distributed versions of the contract, for each value of $N_{threads}$ (so covering both sequential and parallel validators), and for varying numbers of donors. In the transaction sequence $\boldsymbol{T}$ all the donors make their donations, and then claim a refund after the deadline.

Figure 6.7 (left) shows the time (solid lines) and space (dotted lines) used by the sequential validator to process $\boldsymbol{T}$, for the centralized and the distributed versions of the contract. In the figures, we denote by "$1+n$ threads" the runs of the parallel validator with 1 main thread and $N_{threads} = n$ workers. We see that as the number of donors grows, the centralized contract becomes less efficient than the distributed one. This was expected, since replicating the full state in each output takes more time and memory. Since each donation adds an entry in the map, the overall memory consumed is quadratic w.r.t. the number of donors, as confirmed by the figure. By contrast, the distributed contract scales linearly, making it possible to run it with a much larger number of users (up to 1M donors in our experiments, see Table 6.2). Furthermore, the size of transactions in the centralized contract grows linearly, while in the distributed one it is constant. Hence, in eUTXO blockchains the contract would only be usable for a small number of donors. E.g., in Cardano transactions have a 16KB size cap, and so the contract cannot handle more than a few hundred donors. Figure 6.7 (right) shows, for the distributed version of the contract, the time speedup of the parallel validator over the sequential one for varying values of $N_{threads}$. We observe that the speedup grows in the number of users, being only bounded by the number of available threads. This increase in the speedup is coherent with the fact that the percentage of (soft) conflicts, which only depends on the number of users, decreases as this number increases. For instance, for 250 users we have 18.23% of conflicts, and this value drops to 1.64% for 20K users. These conflicts are mostly due to the splitting and merging of map intervals (as discussed at page 108), which occur during the `donate` and `refund` operations.

Map We experiment with a basic map contract featuring only a single rule:

```
contract ConflictMap {
  map (int => int) m;
  inc(i) { m[i] = m[i] + 1; }
}
```

Listing 6.2: The Map contract in hURF.

`inc(i) { m[i]=m[i]+1; }`. The purpose of this experiment is to measure the impact of *soft conflicts* due to data dependencies on performance. Invoking the `inc` rule with distinct values of `i` updates distinct parts of the state, allowing the parallel validator to fully exploit its threads, while invoking the rule multiple times with the same `i` causes a data dependency conflict, preventing the validator to parallelize the dependent invocations. Given any probability value p, we generate the input transaction sequence $\boldsymbol{T}$ so that the probability of causing such a conflict is p, and observe the impact on performance.

The goal here is to study the performance of the distributed Map contract under a varying conflicts probability. Given a wanted probability p of soft conflicts, we construct a sequence of transactions $\boldsymbol{T}$ with $50K$ *random* calls of the rule `inc`, each time determining the actual parameter based on the toss of a biased coin. With probability p, we update `m[0]`; with probability $1-p$, we update `m[i]` where the index i is incremented each time. In this way, we repeatedly update the same map location `m[0]` with probability p, ensuring a soft conflict each time. By contrast, the other map updates access distinct locations `m[i]`, which is very unlikely to cause conflicts. Indeed, in such cases a conflict happens only because of the implicit data dependencies discussed in Section 6.7, when two accesses `m[i]` and `m[j]` need to split the same interval. Since the number of intervals increases after each `m[i]` update, the probability of such conflicts quickly becomes very small, making the effective conflict probability be close to the wanted parameter p.

Figure 6.8 displays the results of our experiment with different validators and values of p. We observe that the parallel validator is generally faster than the sequential one when using at least 2 worker threads. The sequential validator becomes faster only when $p > 60\%$ (dotted vertical line in Figure 6.8), i.e. when the conflict probability is very high. This corresponds to the highly unrealistic scenario where more than half of the transactions update the same map location. When p assumes more realistic values, we observe that the speedup gets fairly close to the number of workers, scaling

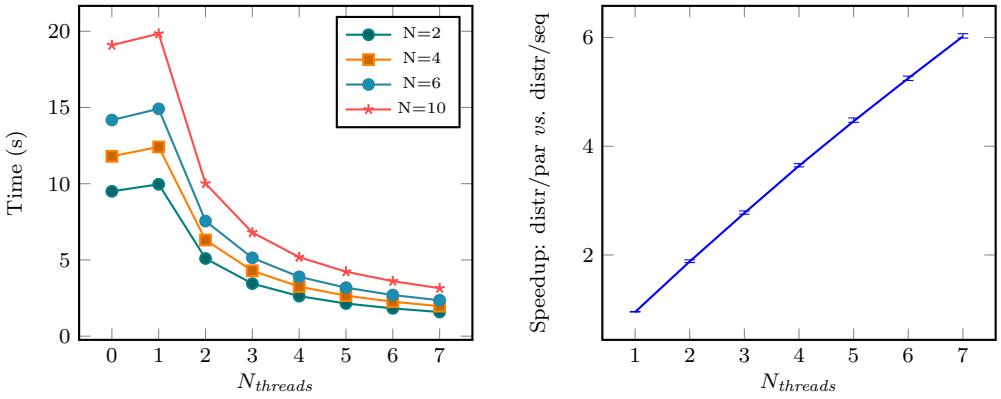


Figure 6.9: Timings and speedup for the Multisig experiments.

linearly with them.

Multisig wallet We consider a wallet contract with three rules: `authorize`, `deposit`, and `withdraw` (see Listing 6.3). The owner authorizes N privileged users, half of which must agree to each withdraw operation by signing it. Instead, any user can deposit funds. We measure the performance of the contract for $N = 2, 4, 6, 10$, so to observe the impact of multiple signature verification. In each experiment, the sequence of transactions $\boldsymbol{T}$ authorizes N users, and then performs $100K$ random operations. When the wallet holds no tokens, we perform a `deposit` operation, otherwise we flip an unbiased coin and perform either a `deposit` or a `withdraw` (signed by $N/2$ users). Validating a transaction also involves verifying an additional signature to pay the transaction fees.

The results are displayed in Figure 6.9. As expected, the average running time roughly scales with the average number of verified signatures. Assuming that the number of deposits is close to the number of withdrawals, the number of verified signatures for N participants is $S(N) = 100K \cdot (0.5 \cdot 2 + 0.5 \cdot (N/2 + 1)) = 25K \cdot N + 150K$. Observing the timings in Figure 6.9 (left) we see that the time for each N is roughly proportional to $S(N)$, confirming that the cost of signature verification is the main factor affecting performance. Figure 6.9 (right) shows the speedups of the parallel validator over the sequential one for a varying number of threads; the error bars show the minimum and maximum speedup over the number $N = 2, 4, 6, 10$ of participants. Note that the error bars are barely visible because these minima and he maxima are very close.

```
contract MultiSig_N {
  const int K = N / 2;
  address owner = "owner_public_key";
  map (address => bool) auth;

  authorize(a) {
    require signedBy(owner);
    auth[a] = true;
  }

  deposit(x) { receive(x:T); }

  withdraw(a1,...,aK,b,x) {
    require(a1 < a2 && ... && aK-1 < aK
      && signedBy(a1) && ... && signedBy(aK)
      && auth[a1] && ... && auth[aK]);
    b.send(x:T);
  }
}
```

Listing 6.3: The Multisig wallet contract in hURF.

From that, we conclude that the speedup does not depend on the value of N but only on the number of threads, confirming that contracts that make heavy use of signature verification can fully exploit parallel validation. Overall, we measured $\sim$1.5% of soft conflicts due to data dependencies. This is not affected in any significant way by the number of users N, nor by the number of threads.

Registry We consider a notarization / registry contract that allows users to register a document and claim its ownership (see Listing 6.4). The contract implements an anti-spoof mechanism based on temporal constraints to prevent adversaries from intercepting registration requests and claim the documents as their own. Users start by calling a rule to record in a map the *hash of the hash* of their document, and a current timestamp. This is needed to prevent frontrunning attacks where an adversary "steals" the actual document hash, and registers it before the legit user. After that, the user calls another rule to reveal the actual hash, and store it in another map. This rule allows the legit user (i.e., the one who was the first to register the double hash) to overwrite any adversaries' claims. Finally, the user who has

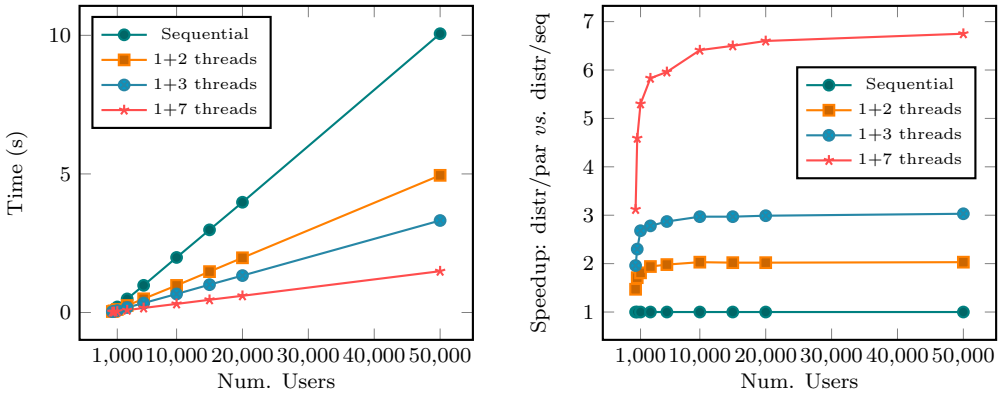


Figure 6.10: Timings and speedup for the Registry experiments.

claimed a hash can permanently record its ownership. The corresponding rule uses time constraints to ensure that the legit user can reveal the hash, as per the second rule. The interest in this contract is in the combination of maps and hash functions; hashes are computed in the second and third rule to obtain the double hash. In each experiment, the input transaction sequence $\boldsymbol{T}$ starts with N distinct users invoking `register`. Users then perform a `claim` on their documents, wait for some deadlines to expire, and finally invoke `own` to finalize their ownership claims. Figure 6.10 displays the timings and speedup for the distributed version of the contract. We observe that the parallel validator is significantly faster than the sequential one. Using $1 + 2$ threads, the speedup is always greater than 1 even for a small number of users (e.g., 1.47 for $N = 250$). Using $1 + 7$ threads, the speedup is always greater than 1, and it tends asymptotically to the bound 7 as the number of users grows. This is due to a low number of soft conflicts ($< 2\%$).

```
contract Registry {
  const int register_accuracy = 10;
  const int claim_time = 100;
  map (hash => int) reserve;
  map (hash => int) priority;
  map (hash => address) owner;

  register(hh,t) {
    require(reserve[hh] == 0
      && validFrom == t
      && validTo == t+register_accuracy);
    reserve[hh] = t;
  }
  claim(h,a) {
    require(reserve[hash(h,a)] != 0
      && (priority[h] == 0 ||
          priority[h] > reserve[hash(h,a)]));
    priority[h] = reserve[hash(h,a)];
  }
  own(h,a) {
    require(validFrom == reserve[hash(h,a)] + register_accuracy +
    claim_time
      && priority[h] == reserve[hash(h,a)]
      && owner[h] == 0);
    owner[h] = a;
  }
}
```

Listing 6.4: The registry contract in hURF.

6.9 Related work

Although the problem of parallelizing transactions is common to all kinds of blockchains, the solutions are radically different depending on whether the blockchain is UTXO-based or account-based. In *UTXO blockchains*, each output may be spent only once: this implies that, when validating a set of possibly conflicting transactions, different orderings will lead to different transactions being included in a block. Therefore, in order to effectively parallelize transactions one needs to decrease the conflict probability, i.e. the probability that their spent inputs are not disjoint. In our approach, we achieved this by distributing the contract state across multiple UTXOs, and by moving the contract balance to an account, rather than keeping it into UTXOs. Instead, in *account-based blockchains*, two transactions targeting the same contract can be executed in any order, but different orders may possibly result in different states (even when both transactions are included in the block). Since validation must be deterministic, parallelizing transactions requires accurately estimating data dependencies to determine whether they can be executed in any order while yielding the same state. This can be done with different techniques, e.g. optimistic execution [9,56], static analysis [22], or by annotating transactions with data dependencies [105].

Furthermore, account-based blockchains can be partitioned into *stateful* when a contract account carries the whole state of the contract besides the contract code, and *stateless* when contract accounts carry no state. Ethereum, Algorand, and Tezos are notable instances of the stateful account-based model, while Solana, Aptos and SUI are instances of the stateless one. The stateful or stateless nature of a blockchain deeply affects the techniques to parallelize transactions and the obtained speedup. We discuss below approaches to parallelize transactions in both models, relating them to ours.

Stateful account-based blockchains One of the first works studying transaction parallelization in the account-based model is [56], which defines a technique for the parallel formation and validation of blocks in Ethereum. Block proposers use speculative execution techniques to find a possible schedule for the parallel execution of the transactions in a block. The obtained schedule is then included in the block, so to allow validators to replicate the same parallel execution when validating the block. The speedup obtained on the proposed (synthetic) benchmark is 1.33x for block formation and 1.69x for block validation, using 3 threads. A main difference between this ap-

proach and ours is that in the UTXO model no explicit schedule for the parallel validation is needed, since transaction inputs implicitly provide the dependencies among transactions. Therefore, our technique does not require to include the schedule as meta-data in blocks, unlike [56]. Another main difference is that, in the account-based model, the state is not explicitly stored within transactions and is computed during the validation process: for this reason, it is impossible to validate a transaction until all its dependencies are validated. Instead, in our hUTXO model, transaction outputs store (parts of) the contract state: consequently, transactions depending on such outputs can have their scripts checked even when the validation of their dependencies is not yet completed (or even started). Dependencies only matter in hUTXO when checking that the contract balance is non negative after each transaction. This check requires to compute the resulting contract balance after the execution of the dependencies: note that such computation is inexpensive, requiring only a few additions per transaction. This in principle allows a theoretical speedup in hUTXO greater than that achievable in account-based models. This is confirmed by our experiments in Section 6.8, where we have achieved a speedup close to the number of available threads.

The Groundhog model [98] proposes a parallelization technique that, rather than searching for a particular ordering of transactions, relies on smart contracts to perform commutative operations. This leads to the same result in any order, so they can be easily parallelized. Since limiting contracts to only use commutative operations would make Groundhog rather inexpressive, a reserve-commit mechanism is exploited to also allow a few selected primitives that might not always commute. These primitives are implemented by first checking a set of constraints during the reserve phase on parallel threads and then applying their effects during the commit phase, ensuring that operations commute. Similar to our model, some burden is placed on the contract code, which must strive to invoke the primitives without violating the constraints, so to allow for parallelization. However, our model is arguably more intuitive: hURF contracts use familiar programming constructs, and ensuring parallelism requires to avoid repeated accesses to the same part of the state. In contrast, Groundhog requires potentially non-commutative contract operations to be restructured as commutative operations subject to constraints, which is less natural.

A few works [22, 92] study the use of static analysis to approximate the portion of the contract state affected by the execution of a transaction, and from this infer a parallel scheduling of transactions. Compared to these

works, parallelizing transactions in our hUTXO model does not rely on static analysis, since the parts of the contract state affected by their execution is already explicit in the inputs of transactions.

The work [22] considers an abstract blockchain model, in order to achieve general results that are applicable both to UTXO-based and account-based blockchains. Assuming a static analysis of the parts of the contract state that are read or written by transactions, they construct an *occurrence net* [37] that describes the possible concurrent executions of transactions, and prove that these executions are semantically equivalent to the sequential ones. Interpreting these occurrence nets as transaction schedules, they show how miners and validators can parallelize the execution of transactions, considering Bitcoin and Ethereum as case studies. Our work refines the results in [22] regarding the parallelization of Bitcoin transactions, developing a technique that makes it possible to parallelize the execution of more expressive contracts, like the ones in Cardano [17].

The work [92] develops a static analysis of smart contracts (written in the Scilla contract language [104]) that allows to distribute transactions across multiple *shards*, i.e. partitions of the whole set of validators [79]. Notably, while most works on sharding focus on simple asset-transfer transactions, [92] was the first one to enable the parallel sharded execution of transactions targeting the *same* contract. To this purpose, the analysis in [92] soundly approximates the effects of a transaction in terms of which part of the state the transaction may write, and whether the writes that affect the same part of state commute. This information is then used to partition the contract state, so that each shard can independently execute (in parallel to the other shards) the transactions that only affect parts of the state assigned to the shard. Similarly to our work, also [92] enables the fine-grained distribution of state, but in the account-based model, while our technique spreads the state across multiple UTXOs. Working on the account-based model allows [92] to perform further optimizations, e.g. executing in parallel commutative operations on the same part of the state. This is not possible in UTXO (as well as in our hUTXO), since each state update requires spending transaction outputs. To make this optimization possible in hUTXO, we could manage the part of the state for which commutative operations are frequent in an account-based fashion, similarly to what we have done for the contract balance. Another key difference is that the sharding technique in [92] requires a static assignment of each part of the contract state to a shard, making that shard the sole executor of transactions that write that parts of the state. Note that this

approach can only cause more conflicts than those occurring in hUTXO. Indeed, while accessing the same state item in different transactions results in a conflict in both approaches, accessing *distinct* items is not a conflict in hUTXO (where these transactions can be parallelized) but could lead to a conflict under static assignment, since distinct items may be mapped to the same shard. In practice, static assignment can give rise to an unbalanced workload between different shards, in case some parts of the state are more used than others.

Stateless account-based blockchains In the stateless model, the contract state is scattered across multiple accounts, which must be supplied as parameters along with method invocations. Therefore, to detect when two transactions targeting the same contract can be executed concurrently it suffices to check if their arguments are disjoint. Based on this, various parallelization techniques are natively implemented in Solana [119], SUI [108] and APTOS [10, 61] and in other custom chains [117].

The key difference between our approach and the parallelization techniques on stateless account-based blockchains is that our parallelization is more fine-grained. To illustrate, assume that the contract state includes a map. In stateless blockchains, this map will be usually stored in a *single* contract account: therefore, if two transactions try to write the map (even at *distinct* keys), they cannot be parallelized. Instead, our fine-grained distribution of the state scatters the map across *multiple* UTXOs: this allows two write operations at distinct keys to be parallelized, with high probability.

UTXO-based blockchains This is the first study to enable the parallel validation of transactions in an UTXO-based model, also considering transactions targeting the same contract. Other papers addressing the scalability problem in the UTXO model devise mechanisms to execute transactions across multiple shards [3, 59]. While these works support cross-shard contract transactions, they cannot parallelize transactions targeting the same contract — so not being able to efficiently process high-throughput contracts. In particular, each of the contracts in Section 6.8 would be executed sequentially by the same shard. The work [85] considers the problem of conflicting transactions, devising an extended UTXO model that performs optimistic updates and tracks the dependencies causing the conflicts. Blockchain consistency is achieved through a conflict resolution mechanism. By contrast, our approach

is conservative: transactions are checked for conflicts before they are sent to worker threads, as discussed in Section 6.8.1. Also, writing different parts of the contract state does not create conflicts.

Representing contract states To represent the state of a contract we first flatten it into a single map Σ. This map is represented by storing the key-value pairs for non-zero values, and open intervals of keys for zero values (the default). Using key-value pais and open intervals is similar to how DNSSEC [67] represents Resource Records (RRs). DNSSEC uses both "positive" RRs to authoritatively resolve a domain name to an address, and "negative" RRs to authoritatively assert that no valid domain name exists within a certain interval. After all the RRs for a domain are signed (possibly off-line), a DNS server can to answer any query within that domain and certify its positive or negative answer. Our approach in similar in that a state output can attest either that Σ holds a given non-zero value at one point (positive information), or that Σ is zero inside an interval (negative information). Unlike DNSSEC RRs, our state outputs are not signed: their authority comes instead from the blockchain and contract scripts invariants.

Contract IDs We exploit contract IDs to ensure contract isolation from maliciously created UTXOs which claim that some state part has a value chosen by the adversary. Thwarting this attack requires constraining the use of contract IDs, both at the blockchain and contract script level. A similar approach is found in Zexe [39] whose aim is the execution of smart contracts over a privacy-preserving UTXO platform (an extension of Zerocash [36]). Contract IDs are also used in Zexe to achieve contract isolation. Zexe is the foundation of the Aleo blockchain which integrates an account-like public state with a UTXO-based private state. The goals of Zexe differ from ours, focusing on the offline private execution of smart contracts over UTXO, instead of the parallel validation of transactions.

Chapter 7

A Simple Optimistic Off-chain Protocol for Bitcoin Contracts

7.1 Introduction

Bitcoin was the first blockchain network, and since its creation it has been mainly used for transferring cryptocurrency in a decentralized, autonomous way. Bitcoin transactions however can do more than basic currency transfers by leveraging the bitcoin scripting language. This language is not Turing-complete, it is loop-free, it only features a few cryptographic primitives, and is in general quite limited in what it can do. In spite of these limitations, Bitcoin still makes it possible to implement many smart contracts. While the expressiveness of Bitcoin as a smart contract platform does not reach the one of blockchains specifically designed for smart contracts (such as Ethereum), the class of contracts that can be implemented in Bitcoin is relatively broad [21, 32]. Some simple contracts can be expressed using a *single* Pay-to-Script-Hash transaction that encodes a suitable spending condition. However, it is also possible to exploit *multiple* transactions to implement more complex Bitcoin contracts, allowing multiple rounds of interaction among participants [6,12,18]. In this work we refer to these multi-transaction protocols as "Bitcoin smart contracts".

Users who want to deploy such a smart contract on Bitcoin can do so with the following standard technique (which we will later detail in Section 7.2.1): first, they generate a tree of transactions that describes all the possible runs of the contract, then they sign all the transactions in the tree, and finally they append to the blockchain only the transactions in a single tree path, according to the participants' choices. While this approach is feasible, it has a main downside: every contract execution step corresponds to appending a transaction from the tree to the blockchain, which requires paying transaction fees. This can be expensive, especially when the number of contract steps is large.

In order to improve fee efficiency, one can attempt to devise methods to perform the execution *off-chain*. Indeed, moving the computation off-chain to save on fees has become a popular means to scale the blockchain smart contracts processing on several platforms [111]. To this aim, a variety of methods has been considered, in particular over account-based blockchain such as Ethereum [52, 72, 76].

In this work, we instead focus on saving fees in Bitcoin smart contracts. We propose a protocol to move most of the execution off-chain, while still guaranteeing the same contract behaviour (Section 7.3). In this protocol participants simulate the contract by exchanging off-chain signatures. In

the best case, in which all participants are honest and follow the protocol correctly, they only need to append three transactions to the blockchain. This does not depend on the number of transactions in the original contract.

Our off-chain protocol has a failsafe mechanism that can be triggered whenever someone detects malicious behaviour. This mechanism moves the contract execution back on-chain, safeguarding the contract from malicious actors. Even in this negative scenario, the steps that were completed off-chain so far are preserved: the on-chain execution starts from the last off-chain state. This reduces the amount of on-chain steps needed to get the contract to completion. When the failsafe is triggered, participants need to pay the fees associated with the remaining contract steps and wait for a few additional time delays. The fees that are saved by the off-chain execution (whether full or partial) can be distributed to the participants when the contract terminates: this serves as an incentive for participants to correctly execute the off-chain protocol.

While our technique is designed to be executed on Bitcoin as-is, it only relies on the fundamentals of the UTxO model, as well as timelocks, a widely adopted primitive in UTxO blockchains. This makes our approach easily adaptable to other UTxO platforms.

Contributions We summarize our contributions as follows:

- We consider on-chain Bitcoin smart contracts, whose execution involves appending a transaction at each step, and we design a optimistic *protocol* to execute such contracts off-chain.

- Our protocol can be run on *stock* Bitcoin — we do not rely on any extensions to the blockchain (e.g., new opcodes, new signature types).

- We study the *security* of our protocol. The off-chain execution of a contract is always faithful to its on-chain one, even in the presence of adversaries. Further, off-chain execution steps are *final*: after a honest participant completes a step, no adversary can cause the contract state to be rolled back to an old one.

- We assess the *efficiency* of our protocol. In the best case scenario, where all participants are honest, only three transactions are needed to execute the whole contract, even when it requires a large number of steps. This drastically reduces the amount of fees that have to be

paid to run a contract. In the worst case scenario, where participants misbehave right after the contract stipulation, our protocol requires to put on the blockchain two additional transactions w.r.t. the on-chain execution. Consequently, fees can only increase by a small amount.

- We showcase our technique through a simple example.

7.1.1 Overview

We consider contract trees C, whose nodes are transactions $\mathsf{T}_0, \mathsf{T}_1, \ldots$. Each transaction redeems its parent in the tree, except for the root T_0 which redeems those outputs used to provide the initial contract funding. Transactions in the leaves terminate the contract, transferring its balance to participants. Tree paths from the root to leaves, represent all the possible contract execution traces, and the branches in internal nodes represent choices that participants take when the contract is executed. Contract trees can be executed on-chain by appending to the blockchain the transactions in a tree path, one by one, paying fees at each step.

We then propose a protocol to move the computation off-chain, saving fees. The key idea is to put an initial transaction Head on-chain, and give all the participants an off-chain copy C_0 of tree C, called a *graft*, that can redeem Head. In principle, any participant can now force the execution to be moved back on-chain by starting to append transactions from C_0. However, participants can also decide which is the next contract step to make $\mathsf{T}_0 \to \mathsf{T}_i$, so identifying the subtree C_i of C_0 which is rooted at T_i. Participants are then given a new graft, a copy of C_i which redeems Head, and they could now move the execution on-chain by appending the transactions from C_i, if they so wish. Participants can instead continue the off-chain execution by agreeing on the next step $\mathsf{T}_i \to \mathsf{T}_j$, and sharing a graft for C_j rooted at T_j. When a leaf is reached, appending its related graft on-chain causes the contract to directly move into its final state without having to pay the fees for each intermediate step.

Our protocol features a few additional mechanisms to protect the off-chain execution from misbehaving participants. In particular, Bitcoin timelocks and an intermediate transaction Init are exploited so to give newer grafts higher priority over older ones and avoid state rollbacks.

7.2 Preliminaries

7.2.1 On-chain contracts

As mentioned in the introduction, we want to consider Bitcoin contract that execute across multiple transactions. We formalize this concept below as a *contract tree*. Intuitively, the nodes of the tree will represent the possible states of the contract, while the edges give a set of conditions that must be satisfied in order to pass from a state to another.

To run Bitcoin contracts on-chain, we follow a technique inspired from [21, 32]. Participants must first follow the *stipulation* protocol (Definition 26), which commits them to actually execute the contract later on. After that, participants must follow the actual *execution* protocol (Definition 27).

Handling fees in Bitcoin contracts Bitcoin requires participants to pay a fee each time a transaction is appended to the blockchain. The price of fees varies according to different factors, such as the congestion of the Bitcoin network and the size of the transaction. The (on chain) execution of Bitcoin smart contracts requires appending multiple transactions to the blockchain, and that needs fees to be paid. To ensure that the contract execution can always progress, we must guarantee that the contract always holds enough funds to pay the required fees. For that reason, we require contract participants to provide enough currency in advance, during the contract *stipulation phase*, so that the contract always can pay fees without needing more funds during the *execution phase*.

During the stipulation phase, we assume participants can reasonably predict the needed fees. For the sake of keeping our presentation simple, in our model we denote with fee a fixed Bitcoin value which is adequate for paying the fees of any contract transaction.

We can now formalize on-chain Bitcoin contract trees as follows.

Definition 23 (Contract tree)**.** A *contract tree* C is a rooted tree of Bitcoin transactions satisfying the requirements below. We distinguish among three types of contract nodes:

- The *root* T_0, that may have any number of inputs, but only has a single output.
- The *inner nodes*, that have a single input and a single output.

- The *leaves*, that have a single input and may have multiple outputs.

Contract transactions must preserve currency: the cumulative value of transaction outputs must be equal to the cumulative value of its inputs, minus the value of a fee.

Except for T_0, the inputs of contract transactions refer to outputs of other contract transactions. Instead, the inputs of T_0 refer to outputs of transactions *outside* C. We require that any such output can only be redeemed with the signature of a participant. We will refer to such participants as the *contract participants*.

The edges of C are determined by the input-output dependencies of the transactions, i.e. there is an edge from T_i to T_j if and only if the input of T_j refers to the output of T_i. Edges are labelled by zero or more *unlock conditions*, which are enforced by transaction scripts (to be explained below). □

In our contract, we allow the following unlocking conditions:

Definition 24 (Unlock conditions)**.** The conditions that can label a contract edge from T_i to T_j are as follows:

- Authorization (labelled sig_{A}): this condition requires contract participant A to sign T_j, authorizing the operation.
- Hash decommitting (labelled rev: S): this requires to reveal the secret S whose hash $H(S)$ was previously committed by a contract participant.
- Waiting (labelled $wait$: t): this condition requires waiting for at least $t \cdot \Delta_w$ seconds after T_i before appending T_j. We fix the time granularity Δ_w to be one hour. We require $t > 0$. □

Our choice of Δ_w is one hour because that is the time needed for a transaction to be confirmed in the Bitcoin blockchain. Conventionally, this happens after six blocks are appended on the blockchain. While different values of the granularity Δ_w could be used, we will assume that Δ_w is large enough to allow contract participants to run off-chain protocols in a timely fashion, with no risk of making $wait$: t expire. More precisely, exchanging protocol messages among participants must take very little time compared to Δ_w. Further, adversaries that aim to delay other participants by being unresponsive

when communicating with them must be detected before a *wait*: t expires. For that, we fix a time $\Delta_r \ll \Delta_w$, and we will require that honest protocol participants detect when another participant whose message is expected takes more than Δ_r to reply. Δ_r must still be large enough to allow honest participants to compute and send their messages.

We now discuss the transaction scripts that must occur in contract trees. These enforce the unlocking conditions on the tree edges.

Definition 25 (Contract transaction scripts)**.** The script of a non-leaf transaction T_i has the form $scr_{j_1} \vee \cdots \vee scr_{j_n}$ where $\mathsf{T}_{j_1}, \cdots \mathsf{T}_{j_n}$ are the children of T_i in the contract tree. Further scr_j performs the following checks:

- All participants must have signed the redeeming transaction, certifying that it is indeed T_j. For this task each participant uses a distinct key for each j. We will refer to these signatures as the *implicit signatures*, since they are required no matter what the unlocking conditions are.
- For each sig_{A} label on the (i, j) edge, the participant A must have provided an additional signature with a different key from the above ones.
- For each *rev*: S label, a preimage of $H(S)$ must be included among the witnesses of T_j, effectively making S public.
- For each *wait*: t label, T_j must include a relative timelock that prevents it to be appended to the blockchain before t units of time since T_i. □

An example of a contract tree C is depicted in Figure 7.1. There, we show a few "deposit" transactions which are already on the blockchain, highlighted in yellow, which are not part of C. The root T_0 of C refers to such transactions as its inputs, as denoted with the dotted lines, which we label with the signatures which are required to spend them. $\mathsf{A}, \mathsf{B}, \cdots$ denote the contract participants.

The rest of the transactions form the actual contract tree C. We represent the edges of C with solid arrows, corresponding to the scripts as defined in Definition 25. We label the solid lines with their unlocking conditions. We remark that transaction scripts enable such edges only when the condition is satisfied, but they also require an *implicit* signature by each contract participant. In all our figures, we do not show such implicit signatures on solid lines so to avoid clutter.

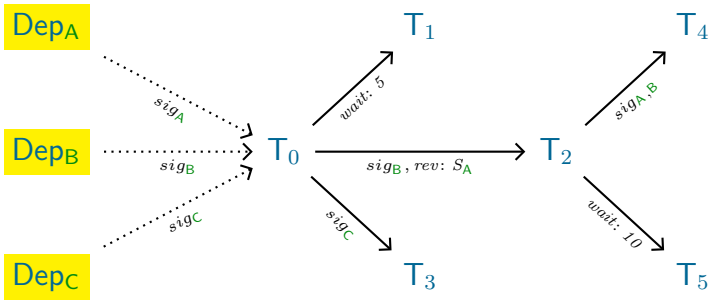


Figure 7.1: Example of a contract tree. sig_{P} labels require an authorization by P (signature on the redeeming transaction). rev: S_{A} labels require revealing the secret S_{A} whose hash was previously committed. $wait$: t labels require waiting for at least $t \cdot \Delta_w$ time.

On contract leaves We remark that or definition of contract tree essentially poses no requirement on the outputs of the contract leaves. Typically, leaves will distribute the contract balance in some way among participants, but they can also transfer Bitcoins to other parties, or even require complex redeeming conditions. For instance, in the example above, the contract is designed so that its leaves transfer the bets to the winning participant. More in general, the actual script used in contract leaves is mostly immaterial to our results. However, as we will see in Section 7.2.4, it is possible to design leaves so to redistribute the unspent transaction fees which arise from executing a short contract path. This will be taken into account when assessing the cost of executing a contract (both in the on-chain and off-chain cases).

7.2.2 Stipulation and Execution of On-chain Contracts

Stipulation Contract stipulation involves the following steps:

Definition 26 (On-chain stipulation protocol)**.**

1. Participants must first communicate their interest in stipulating C, making it known to everyone.

2. Each participant then generates all the transactions in the contract tree, except for their witnesses.

3. Then, participants exchange all the *implicit* signatures (as per Definition 25) on each transaction. If an invalid implicit signature is received, the stipulation is aborted.[1]

4. Participants exchange the signatures for the root transaction T_0, so that it can redeem its input.

5. T_0 is put on the blockchain. The contract is now *stipulated*. □

The last item, appending T_0, is the "point of no return". Stipulation could be aborted at any time before that, but once T_0 is on the blockchain the contract must now be executed.

An important aspect of the stipulation protocol is that all the implicit signatures in the contract tree are exchanged and verified before T_0 is signed. If we allowed T_0 to be signed earlier, it would be possible to put it on the blockchain and start the execution of the contract before all the implicit signatures are made, allowing an adversary to deny some of their implicit signature and effectively stall the execution of the contract, freezing its funds forever.

Execution Once the stipulation protocol is concluded and T_0 is on the blockchain, participants can execute the contract by suitably appending the contract transactions along any tree path, as long as the unlocking conditions in that path are satisfied. This is formalized by the following protocol:

Definition 27 (On-chain execution protocol)**.**

1. Let T_* be the single transaction of C which is on the blockchain with its outputs still unspent.

2. If T_* is a leaf of C, the execution of the contract is over.

3. If T_* is a non-leaf, we consider its children. Participants may choose to reveal their secrets, to exchange their signatures on some child transactions, or to wait, according to their wishes. In this way, they can enable the transition from T_* to the children they wish to put on the blockchain.

[1] We stress that these implicit signatures do not include those needed to satisfy the authorizations in the unlock conditions. Those will be only be made during the contract execution.

4. Any participant can then choose one of the enabled children T_j and append it to the blockchain, redeeming T_* in the process. We stress that the script of T_* is satisfied: all the implicit signatures have already been exchanged in the stipulation phase, while the unlocking conditions have been satisfied in the previous step (Item 3).

5. After T_j has been appended to the blockchain, we restart the protocol from the beginning. □

Note that in Item 3 the protocol does *not* force participants to reveal their secrets or exchange signatures. Rather, honest participants can freely choose whether to do so. For instance, if a contract tree node has two children and both require an authorization form participant A, then the contract effectively grants A to choose, at execution time, which contract subtree must be executed. Here, A could authorize either branch, both, or none.

Further, we remark that in Item 4 there might be multiple children that can be appended on the blockchain. If that happens, multiple (honest) participants can attempt appending different transactions on the blockchain, all spending T_*. This causes a race: the Bitcoin network will effectively choose which transaction is appended, and which is discarded. This is not an issue on its own. If undesired, the contract can be designed to reduce such races by carefully using the unlocking conditions, including temporal constraints, so that one of the options becomes enabled before the other, making it possible to avoid the race.

We remark that the execution protocol only requires a single participant to be honest. Indeed, the presence of a single honest participant is enough to guarantee that the contract will be executed according to one of its paths. This is because each non-root node requires an implicit signature by *all* participants, and a honest participant does not reuse the key they used in the stipulation protocol to sign new transactions. Consequently, only the stipulated paths can be followed.

Further, the execution protocol does not require participants to interact except when a participant wants to satisfy an unlocking condition, and has to reveal secrets or share new signatures. A single honest participant is still able to make the contract progress on its own, without any help from others, as long as they choose a path whose unlocking conditions can be satisfied by them. Indeed, as already remarked in Item 4, the implicit signatures are already known by the honest participant. The contract execution can therefore

only stop when reaching a contract leaf, therefore exiting the contract itself, or when reaching an internal node whose children all require authorizations or secrets which are being refused. The last case can be avoided by designing the contract so that each non leaf always has a children requiring at most a *wait*: t condition: in this way a leaf can always be reached.

7.2.3 Example: On-chain Best-of-3 Bet

We illustrate the on-chain execution of a contract through an example. The tree of transactions presented in Figure 7.2 implements a best-of-three bet between Alice and Bob, on some kind of recurring match between two teams. Before the competition starts, an oracle commits the hashes of six secret messages: three of them denote that the first team won a match, while the other three denote that it lost[2]. After each match the oracle certifies the victory or loss of the team by revealing the suitable secret message. These secrets can then be used by Alice and Bob to update the state of the contract, appending a new transaction to the blockchain. Transactions in the contract are named to represent the win-loss record associated to the corresponding state, from the point of view of the first team, meaning that W?? represents the state in which the first team has won once, and two matches still have to be played, while LWL represents the state in which the first team has lost, then won, then lost again. The contract ends when either team has won two out of three matches, or when both Alice and Bob agree to an early payout, terminating the bet before the three matches have been played. The contract has 10 possible terminations, shown in the tree as 10 leaves.

The transactions WLW, WW transfers all the balance of the contract to Alice, who bet on the first team winning. Likewise, the transactions LL, LWL transfer the prize to Bob. The early payout option, that can be taken only if Alice and Bob both agree on it, is performed by leaves Out_W (which gives a bigger share to Alice, since it can be taken after the first win has been revealed), Out_{WL} and Out_{LW} (which split the prize equally), and Out_L (which pays a bigger share to Bob).

It is important to notice that in this contract the closer a leaf is to the root, the more total money it will have available, due to having to pay less

[2]The 6 messages contain the strings $W_1, W_2, W_3, L_1, L_2, L_3$ and are padded with a nonce for security. Note that the oracle is not a contract participant, and does not need to be aware of the bet between Alice and Bob. Indeed its role could be performed by three distinct oracles, one for each match.

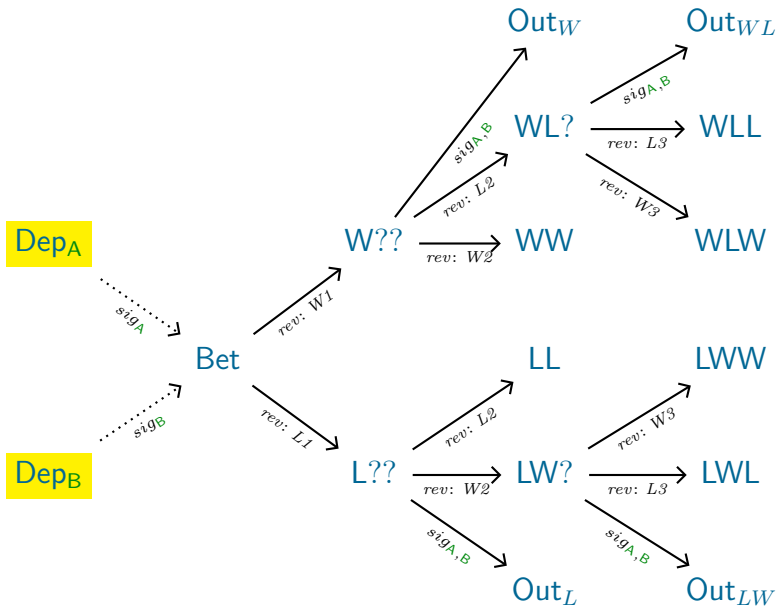


Figure 7.2: The Best-of-3 bet contract tree crafted during stipulation. Only the deposit transactions are on the blockchain at this time (highlighted in yellow). The stipulation phase completes by putting Bet on chain, after which execution can start.

in fees. For this reason, the transactions WW, LL, Out_W, Out_L can pay back to Alice and Bob a part of the fees that they anticipated. In such a small contract, the difference is only slight, but in deeper contracts, i.e. a best-of-5 bet, they start to become more impactful. Figure 7.3 shows a possible on-chain execution of the contract.

7.2.4 More on fees

For the sake of simplicity, in Definition 23 we used a fixed amount of currency (fee) for the fees of any transaction. We remark that this simplification is not strictly necessary, and that different fees can be used throughout the contract tree, e.g., making them to be proportional to the transaction size. We still require that the inputs on any tree node are sufficient to pay for the fees, hence the contract root T_0 must obtain from its inputs enough currency to pay for all the needed fees in any of its execution paths.

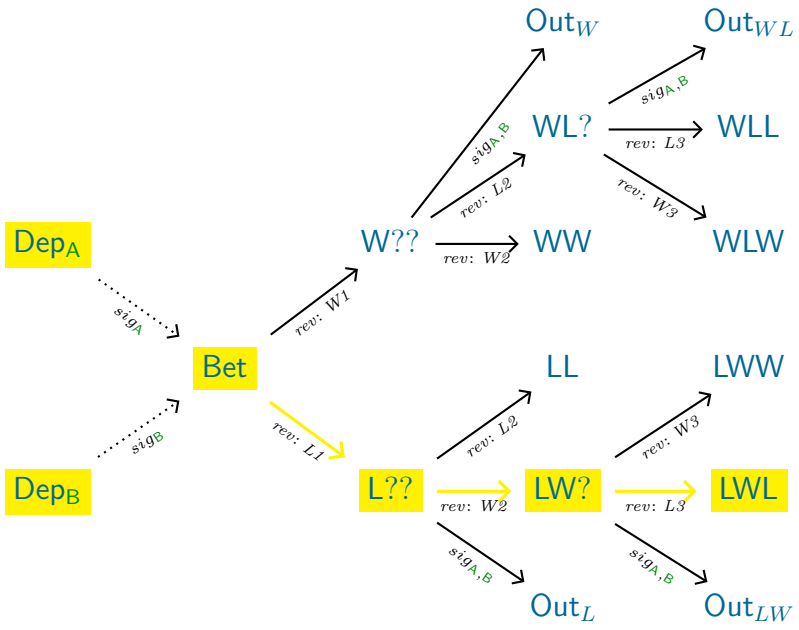


Figure 7.3: A possible contract execution path. After the contract is fully executed, all the transactions highlighted in yellow are on the blockchain, while the others became invalid and must be kept off-chain (or deleted). To perform the steps in the path, participants must wait for the oracle to decommit one of the hashes, so that they exploit the corresponding preimage to redeem the previous transaction in the path with the next one, so moving to the next contract state.

Note that different tree execution paths might use a different amount of fees, whether because the individual transactions fees are different, or simply because a tree path is shorter, hence it requires fewer execution steps. Because of this, the contract balance which becomes available to tree leaves can vary. While not strictly mandatory, in most common contracts the root transaction requires all the contract participants to equally share the fees cost. Dually, the contract leaves refund the unspent fees back to the participants. In the cheaper execution paths, the unspent fees can be significant, while in the most expensive paths there are no unspent fees to refund.

7.3 Off-chain contracts

In this section we present a technique to move the contract *off-chain*. For this, we use new off-chain stipulation and execution protocols. Before describing them in a general way (Definitions 28 and 29), we show them in the specific case of the best-of-three bet example of Section 7.2.3.

7.3.1 Example: off-chain best-of-three bet

We now walk through the steps that are needed to move off-chain the best-of-three bet. First, Alice and Bob generate a slightly modified copy of the contract (see Figure 7.4a), where there is a new Head transaction that has as inputs the two initial deposits, an Init transaction (whose input is Head), and a modified Bet transaction, which now features a timelock of $3\Delta_w$ relative to its input, Init. From there, the modified contract follows the structure of the original one.

Again, mirroring the protocol described for the on-chain stipulation of the contract, Alice and Bob send each other the implicit signatures needed for every transaction, leaving for last the ones that unlock the two deposits and enable Head. Once they have all the needed signatures, they append Head to the blockchain, spending their deposits. Now the on-chain transactions are exactly those highlighted in Figure 7.4a. Notice that, at this point, the transaction Init is enabled (since they had already exchanged the implicit signatures) but, unlike in the on-chain contract execution, Alice and Bob should refrain from appending it to the blockchain right away. Indeed, appending Init would interrupt the off-chain execution of the contract, forcing it to be executed on-chain, and causing Alice and Bob to pay the fees at

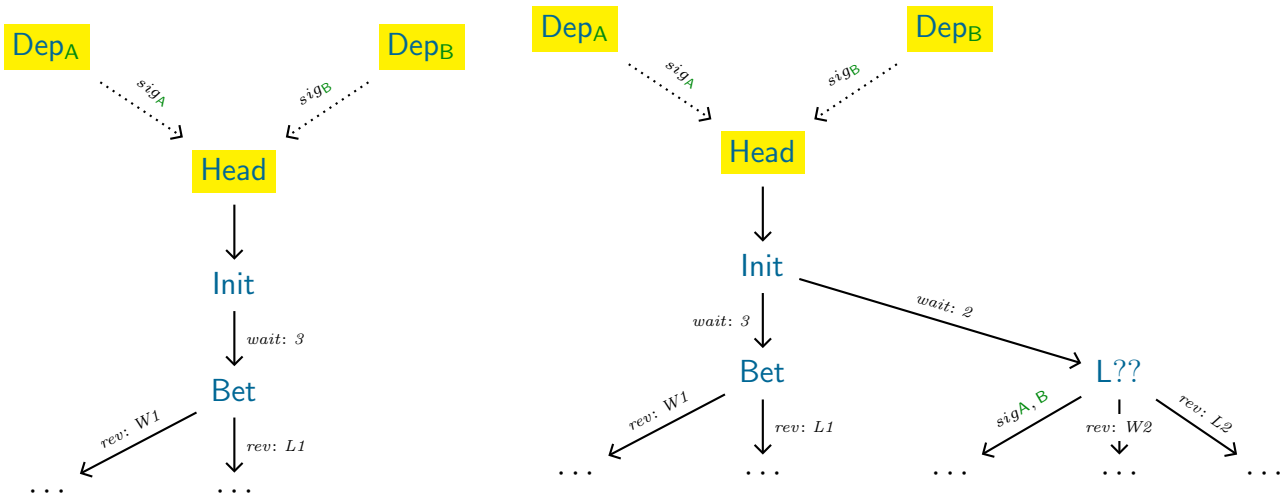
each step. The ideal scenario is the one in which Init is appended only after the off-chain execution is completed (i.e. a contract leaf is reached). The fact that Init is enabled from the start is an intended feature of the protocol: it is, in fact, a failsafe mechanism, that allows one participant to force the on-chain execution whenever the other is behaving maliciously and refusing to cooperate with the off-chain protocol.

From now on, our example will follow a specific execution trace of the contract, the one shown in Figure 7.3, assuming that the oracle will reveal "L1", "W2", and "L3", and that neither Alice nor Bob will agree to take an early payout. After the oracle reveals "L1", in the original on-chain contract Bob would append L?? to the blockchain moving the state forward. Instead, in the off-chain protocol, the two participants create a *graft*: a copy of the subtree rooted at L??, modifying L?? so that its input is now the Init transaction, and so that it has a relative timelock of $2\Delta_w$. Then, they graft this subtree to the off-chain contract (see Figure 7.4b) by exchanging the implicit signatures needed by the transactions in the subtree. Completing this graft effectively makes the off-chain contract perform a step to the new L?? state.

Note that, if at this point either participant wanted to bring the contract execution on-chain, then they could do so by appending the Init transaction, waiting for $2\Delta_w$ units of time, and then appending L??, carrying on with the on-chain execution from that point onward. We remark that, after Init has been appended to the blockchain, an adversary could attempt to "roll back" the contract to a previous state, by appending Bet instead of L??. However, due to the delays enforced by the timelocks, any honest participant is always able to prevent this by appending L?? *first*. Indeed, L?? unlocks one Δ_w *earlier* than Bet.

The next off-chain steps are performed in a similar fashion: Alice and Bob can graft a copy of the subtree of transactions rooted in the one that represents the next state, modifying it so that the root has as input the transaction Init, and so that it now includes a timelock which is crucially smaller than the ones created in the previous steps. By using a smaller timelock, this graft is given a higher priority w.r.t. the previously created grafts. After the graft is created, they exchange the implicit signatures, leaving for last the ones for the subtree root.

Assuming now that Alice and Bob are continuing the off-chain execution, they will wait for the oracle to reveal the next result (in this case "W2"), then graft the subtree rooted at LW?. Finally, after "L3" is revealed, they



(a) Start of the off-chain tree

(b) After the first step, a subtree is grafted

Figure 7.4

can graft the subtree consisting only of the transaction LWL, getting to the full off-chain tree shown in Figure 7.5. Now that the contract has no further possible steps, Alice and Bob complete the execution by putting both Init and LWL on-chain. In this way, the contract is completed with only 3 on-chain transactions (Head, Init, and LWL), when the on-chain protocol required 4 transactions for the same execution path. Consequently, the participants have saved one fee, which can be redistributed by LWL.

7.3.2 Stipulation and Execution of Off-chain Contracts

Moving away from the example, we now present the general off-chain stipulation and execution protocol for an arbitrary contract. In the rest of this section, we denote with C an on-chain contract tree that we wish to execute off-chain.

The fields of all the transactions created in this protocol are shown in Figures 7.6 and 7.7 and discussed in Section 7.3.3.

Stipulation The off-chain stipulation protocol is similar to the on-chain one, but involves two more transactions (Head and Init). Intuitively, the Head

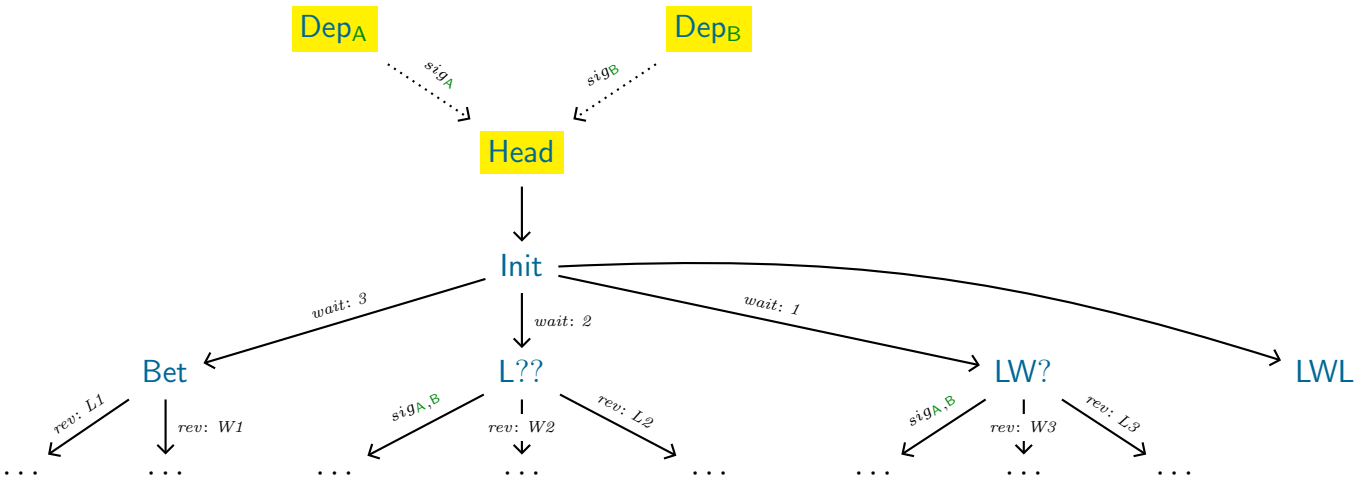


Figure 7.5: Complete off-chain contract

transaction is used to make participants commit to executing the contract. Instead, the Init transaction acts as a failsafe mechanism against malicious behaviour, allowing any participant to move the contract execution from off-chain to on-chain, at any time.

Definition 28 (Off-chain stipulation protocol)**.**

1. Create a transaction Head, making its inputs provide the initial contract funds, similarly to the inputs of the root of C, but with two additional fees. Contract participants must provide for all these funds, as in the on-chain stipulation protocol.

2. Create a transaction Init redeeming Head.

3. Create a copy C_0 of the contract tree C, and modify it so that the root of the tree now redeems Init instead of the outputs external to the contract. Redeeming Init in this way must require waiting for the depth of C_0 times Δ_w: this is implemented in Bitcoin adding a relative locktime to the root of C_0.

4. Make the scripts of each non-leaf transaction mentioned above require a signature from every contract participant in order to be redeemed. Such signatures are made with a single-use key, much like the implicit

signatures of the on-chain contract. We will therefore also refer to these signatures as the *implicit* ones. The scripts in C_0 still check the unlocking conditions as in the original C.

5. Exchange the implicit signatures for every transaction. (Note that this does not involve signing Head.)

6. Sign Head and append it to the blockchain, redeeming its inputs. After that, the contract becomes *stipulated*. □

We stress that, after Head is on the blockchain, the contract could be executed on-chain by any participant. To do so, it suffices to (i) redeem Head with Init, (ii) redeem Init (after the timelock expired) with the root of C_0, and finally (iii) continue from these with the on-chain contract execution. This is possible because, after stipulation, all the implicit signatures are known by all participants.

Now that participants have established this baseline security guarantee, they can attempt to execute the contract off-chain according to the following optimistic protocol.

Definition 29 (Off-chain execution protocol)**.**

1. Let C_* be a variable referring to a copy of a subtree of C. Initially, we let C_* be C_0, the copy of C generated during the stipulation phase.

   Intuitively, C_* represents the current off-chain contract state.

2. Run the *grafting protocol* in Definition 30 repeatedly, aborting it immediately as soon as one of the following conditions is met:

   (a) The transaction Init is appended to the blockchain. (This can only happen because of a dishonest participant.)

   (b) A (dishonest) participant fails to cooperate, i.e. some grafting protocol message is expected from them but it is not received within time Δ_r.

   (c) C_* becomes a tree containing only one leaf, completing the off-chain execution of the contract.

3. Append Init to the blockchain (if not already there).

4. Append the root of the latest graft C_* as soon as its timelock expires.

5. If C_* comprises more nodes than just a leaf, continue the execution of the contract on-chain, appending the transactions of C_*, following the on-chain execution protocol of Definition 27. □

The above protocol repeatedly creates grafts C_* in Item 2, effectively advancing the current off-chain contract state, one step per graft. To do so, it uses the grafting protocol (to be defined below). Creating a new graft, however, requires the cooperation of all the contract participants, and as such it is subject to some threats from adversarial participants. We will study such threats in Section 7.3.4. Here, we only remark that the off-chain execution protocol defends itself from such threats by detecting them in Item 2, and then by moving the execution of the contract on-chain (from Item 3), as a failsafe mechanism. This is possible since, after each graft is created, participants have the option to append Init to the blockchain, wait for the root of the latest graft C_* to become enabled, and append it, so that the contract can be continued from there.

We now detail the protocol to create the next graft.

Definition 30 (Grafting protocol)**.** In the following, we let T_* be the root of C_*, and $\mathsf{T}_1, \cdots, \mathsf{T}_k$ be its children.

1. Each participant share with the others their wishes about which child (or children) among $\mathsf{T}_1, \cdots, \mathsf{T}_k$ should be the next contract state[3]. Participants must then reach an agreement on a single child T_i. If that is not possible because of conflicting wishes, the protocol is aborted. (This makes the execution protocol continue from Item 3.)

2. Let C_i be a copy of the subtree rooted in T_i, modified as follows:

   - The root of C_i has as its input the Init transaction (instead of T_*).
   - The root of C_i uses a relative timelock that requires waiting for a time equal to the depth of C_i times Δ_w.
   - The values of all the non-leaf outputs of C_i are increased by fee to account for the saved fees due to the step $\mathsf{T}_* \to \mathsf{T}_i$ being now only performed off-chain.

[3] A participant should not agree to child T_j when they are not intending to reveal the secrets or provide the signatures to satisfy the unlocking conditions for the step $\mathsf{T}_* \to \mathsf{T}_j$. Doing so will cause the grafting protocol to abort later on, eventually causing the contract execution to be moved on-chain. Even in such case, the security of the contract is not compromised.

- The values of the outputs of each leaf transaction of C_i are increased, refunding the participants with the saved fee. [4]
- The keys used in the scripts in C_i for verifying the implicit signatures are substituted with fresh keys, so that they keys are still single-use (they are only used to sign a single transaction), similarly to the procedure used for creating C_0 in the stipulation phase.

3. Participants must now satisfy the unlocking conditions of the $\mathsf{T}_* \to \mathsf{T}_i$ edge. This includes waiting, revealing secrets, and sending signatures. The witnesses of (the copy of) T_i are adapted accordingly.

4. Participants exchange all the implicit signatures for the transactions in C_i. The signatures for the root are exchanged last. After the exchange is completed, we say that C_i becomes *grafted*.

5. We set the current off-chain state C_* to be C_i. □

We remark that the signatures generated in Item 4 are indeed needed, and the previously generated signatures cannot be reused. This is because the transactions in the grafts have different input fields from the ones in the original tree, and signatures must be calculated on the whole transaction. Like in the on-chain contract protocol, we emphasize that signing the whole graft before its root is essential to avoid stalling attacks. For this reason we can not sign the graft transactions step-by-step after the graft root is already on-chain: at that time all the implicit signatures must already be known by participants.

7.3.3 Detailed transaction format

The exact format of the transactions created by our off-chain protocol are shown in Figures 7.6 and 7.7. The notation for the Bitcoin transaction in the figure is taken from [13].

Our transactions involve a number of distinct cryptographic keys, which must be exchanged by the participants. More in detail, we assume that each participant A has a "master" signing key pair whose public part is shared

[4] We do not specify the exact redistribution method, since it is immaterial to our protocol. Participants may choose any redistribution method as long as they agree on it.

Head
in: $\mathsf{Dep_A}.\mathsf{out}(0)$, $\mathsf{Dep_B}.\mathsf{out}(0)$, $\cdots$ wit: $sig_{\mathsf{A}}, sig_{\mathsf{B}}, \cdots$ out(0): $(\lambda\varsigma.\mathsf{versig}(\mathsf{AHead}, \varsigma_1) \wedge \mathsf{versig}(\mathsf{BHead}, \varsigma_2) \wedge \cdots, v\text{Ƀ})$

Init
in: $\mathsf{Head}.\mathsf{out}(0)$ wit: $sig_{\mathsf{AHead}}, sig_{\mathsf{BHead}}, \cdots$ out(0): $(\lambda\varsigma.\mathsf{versig}(\mathsf{AInit}, \varsigma_1) \wedge \mathsf{versig}(\mathsf{BInit}, \varsigma_2) \wedge \cdots, v'\text{Ƀ})$

Figure 7.6: Head and Init transactions

Root $\mathsf{T}_{i,i}$ of graft C_i, copy of T_i with children $\mathsf{T}_{j_1}, \mathsf{T}_{j_2}, \ldots$
in: $\mathsf{Init}.\mathsf{out}(0)$ relLock: $\mathsf{depth}(C_i) \cdot \Delta_w$ wit: $sig_{\mathsf{AInit}}, sig_{\mathsf{BInit}}, \cdots$ out(0): $(\lambda\varsigma. \bigvee_k \left(\begin{array}{l} \mathsf{versig}(\mathsf{A}(i,j_k), \varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(i,j_k), \varsigma_2) \wedge \cdots \wedge \\ \text{unlocking conditions for } \mathsf{T}_i \to \mathsf{T}_{j_k} \text{ without implicit versigs} \end{array} \right), v''\text{Ƀ})$

Internal node $\mathsf{T}_{i,j}$ of graft C_i, copy of T_j child of T_p
in: $\mathsf{T}_{i,p}.\mathsf{out}(0)$ wit: $sig_{\mathsf{A}(i,j)}, sig_{\mathsf{B}(i,j)}, \cdots$ out(0): analogous to the graft root output

Leaf node $\mathsf{T}_{i,j}$ of graft C_i, copy of T_j child of T_p
in: $\mathsf{T}_{i,p}.\mathsf{out}(0)$ wit: $sig_{\mathsf{A}(i,j)}, sig_{\mathsf{B}(i,j)}, \cdots$ out(0): $(\lambda\varsigma.\mathsf{versig}(\mathsf{A}, \varsigma_1), v_{\mathsf{A},j}\text{Ƀ})$ out(1): $(\lambda\varsigma.\mathsf{versig}(\mathsf{B}, \varsigma_1), v_{\mathsf{B},j}\text{Ƀ})$ ⋮ ⋮ ⋮ ⋮

Figure 7.7: Graft transactions

Graft root $\mathsf{T}_{Bet,Bet}$, copy of Bet

in:	Init.out(0)
relLock:	$3\Delta_w$
wit:	$sig_{\mathsf{AInit}}, sig_{\mathsf{BInit}}$
out(0):	$(\lambda\varsigma x.(\mathsf{versig}(\mathsf{A}(\mathsf{Bet},\mathsf{L??}),\varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(\mathsf{Bet},\mathsf{L??}),\varsigma_2) \wedge H(x) = H(L1)) \vee$ $(\mathsf{versig}(\mathsf{A}(\mathsf{Bet},\mathsf{W??}),\varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(\mathsf{Bet},\mathsf{W??}),\varsigma_2) \wedge H(x) = H(W1)), v\text{฿})$

Graft internal node $\mathsf{T}_{Bet,L??}$, copy of L??

in:	Bet.out(0)
wit:	$sig_{\mathsf{A}(\mathsf{Bet},\mathsf{L??})}, sig_{\mathsf{A}(\mathsf{Bet},\mathsf{L??})}, L1$
out(0):	$(\lambda\varsigma x.(\mathsf{versig}(\mathsf{A}(\mathsf{Bet},\mathsf{LL?}),\varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(\mathsf{Bet},\mathsf{LL?}),\varsigma_2) \wedge H(x) = H(L2)) \vee$ $(\mathsf{versig}(\mathsf{A}(\mathsf{Bet},\mathsf{LW?}),\varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(\mathsf{Bet},\mathsf{LW?}),\varsigma_2) \wedge H(x) = H(W2)) \vee$ $(\mathsf{versig}(\mathsf{A}(\mathsf{Bet},\mathsf{Out}_L),\varsigma_1) \wedge \mathsf{versig}(\mathsf{B}(\mathsf{Bet},\mathsf{Out}_L),\varsigma_2) \wedge \mathsf{versig}(\mathsf{A},\varsigma_3) \wedge \mathsf{versig}(\mathsf{B},\varsigma_4)), v'\text{฿})$

Figure 7.8: Example of graft transactions.

with other participants before contract stipulation starts. Each message from A is then signed with that master key so to ensure its authenticity. In particular, when A needs to generate new key pairs to sign certain transactions, A shares the public key with the other participants, using the master key to certify that the public key is indeed A's key and that its purpose is to sign a given transaction. As shown in the figures, each participant A needs to generate key pairs AHead and AInit for the Head and Init transactions. Further, each time a graft C_i is created, copying transaction T_i from the original contract tree and its descendants T_j, each participant generates a key $\mathsf{A}(i,j)$ for each transaction $\mathsf{T}_{i,j}$ in the graft. Finally, we assume that each participant own a key to redeem basic deposits of Bitcoins. This key, which we simply denote with A, is only used at contract stipulation (in Head) and at contract termination (in graft leaves).

We now comment on the transactions and their fields. The Head transaction spends a few deposits owned by the participants to provide the initial contract balance v฿, as shown in the input field in. The witness field wit of the transaction carries the needed signatures. These witnesses can be segregated away according to the SegWit Bitcoin mechanism [78], if so wished. The single output out involves a script that takes a few signatures ς and verifies them with the public parts of the AHead, ... key pairs.

Transaction Init redeems Head in a similar way. Note that it involves its own set of keys. Its output value v'฿ is the value v฿ in Head, minus a

transaction fee.

Graft transactions are more complex. In the figures we show the format of the transactions corresponding to the root, internal nodes, and leaf nodes of the graft. The root redeems Init, and uses the relative locktime field relLock to require that $\mathsf{depth}(C_i) \cdot \Delta_w$ time has passed since Init was appended on the blockchain. The witnesses provide the required signatures. The script accounts for all possible children in the graft: its form is a logical disjunction of the conditions required for each move. Indeed, for each possible move $\mathsf{T}_i \to \mathsf{T}_{j_k}$ we require the implicit signatures made with keys $\mathsf{A}(i, j_k), \ldots$, as well as the other specific unlocking conditions for that move (as per Definition 25). The script for the off-chain graft transaction $\mathsf{T}_{i,i}$ is therefore the same as the one used in the on-chain transaction T_i, except that the implicit signatures are made with distinct keys each time a graft is created. The output value v''฿ is the value v'฿ in Init, minus a transaction fee.

The transactions for internal graft nodes are similar to the root, except for the following differences. First, they redeem their parent in the graft instead of Init. Second, they are signed with different keys ($\mathsf{A}(i, j)$ instead of $\mathsf{A}Init$). Their output is completely analogous to the one for the graft root, and its value is once again the same as the value of the parent minus a transaction fee. Note that internal nodes can optionally involve a relLock:t field when there is a *wait*: t unlocking condition from their parent. (By contrast, the root transition in the graft does not need to account for its own *wait*: t since the waiting is to be performed off-chain. Its relLock field is instead used to ensure it has the right priority.)

Finally, transactions for leaf nodes simply redeem their parent and redistribute the contract balance to participants. This includes both the amount which would have been distributed by the on-chain contract *and* the additional savings from the avoided fees. Recall from Section 7.2.4 that, even in the on-chain execution, at the end of the contract the balance can be different depending on the length of the tree path that was executed. When the contract is stipulated, the initial balance must account for all possible execution paths, their fees at each step, and their intended transfers to the participants once the contract ends. Since longer paths require more fees to be paid, shorter paths can be reached with a larger balance than desired, in which case the contract designer can choose to redistributed the unspent fees to the participants.

In the off-chain contract execution, this phenomenon is also present: if a graft C_i is used to move the computation on-chain, in the general case we

still have paths of different lengths requires different fees. On top of the fees which are saved because we took a shorter path, we have the savings coming from the off-chain execution: no fees have been spent to move from the contract tree root to the graft root. Effectively, the number of the additional saved fees due to off-chain execution is $\mathsf{depth}(C) - \mathsf{depth}(C_i) - 2$, where the number 2 accounts for the Head and Init transactions. In Figure 7.7, the values in the leaf outputs $v_{\mathsf{A},j}$₿ sum to the contract balance in the leaf, and comprise both the intended end-of-contract transfers and the redistributed unspent fees.

As a concrete example, we show in Figure 7.8 a few transactions from the Bet contract in Section 7.2.3. These are from C_{Bet}, the first graft created by the protocol, so its root is a copy of the Bet transaction which is the root of the (on-chain) contract tree C. The graft root $\mathsf{T}_{Bet,Bet}$ redeems Init and its script allows moves towards L?? and W??, depending on which hash preimage x has been revealed. The relative timelock is $3 \cdot \Delta_w$ since the depth of the graft is the same as the one of the whole C contract, i.e., we need at most three moves to reach a leaf. The second transaction in the figure is internal node $\mathsf{T}_{Bet,L??}$ which redeems $\mathsf{T}_{Bet,Bet}$ using the appropriate signatures and the $L1$ preimage. The script used in the output is a three-way disjunction. The first two cases allow moving towards LL? or LW?, depending on which new preimage x is revealed at this time. Instead, the last case allows moving towards Out_L: the contract requires that this step is authorized at execution time by both participants. Therefore, as already done in the on-chain contract tree, the script requires both implicit signatures (which are exchanged at stipulation time), and additional authorization signatures (to be provided during execution)[5].

7.3.4 Threat analysis

We now study the potential security threats that can affect the off-chain execution protocol. Intuitively, the safety of our off-chain protocol is predicated on three properties.

- First, we want any off-chain execution of a contract to be coherent with an on-chain execution of the same contract. More in detail, each

[5]This implementation would be optimized to require only the two execution-time signatures, as done in [32]. Here we use distinct signatures to make the purpose of each signature more evident.

off-chain execution step must have an on-chain counterpart.

- Second, we want to ensure that any on-chain execution of a contract can be also performed off-chain. More precisely, whenever participants can take an on-chain step from a state, the off-chain execution protocol must allow them to take a corresponding step from the corresponding state. In particular, this ensures that the off-chain execution never stalls the contract except in those cases where the on-chain execution already stalls.

- Finally, we want our off-chain execution to be *final*: if the off-chain protocol reaches a certain state, then the adversary must not be able to "roll-back" the contract to a previous state and continue the execution from there.

In our threat analysis we will postulate the existence of at least one honest contract participant, who is assumed to be interested in having the contract proceed exactly as stipulated.

Off-chain behavior can not exceed the on-chain one The off-chain execution protocol proceeds by generating and signing grafts, in this way updating the current off-chain contract state. By construction, honest participants sign a new graft only when the original on-chain contract allows a step between the old and new contract state and the associated unlocking conditions are met, hence no adversary can cause grafts to be produced unless following the original contract semantics.

An adversary can also attempt to disrupt the contract execution by first appending Init and the root of a graft, so moving the contract execution on-chain, and then continue its attack from there. However, every transaction in the graft requires a signature from every participant, and one of them is honest, so the adversary can not redeem (non-leaf) graft transactions except by using other graft transactions which are linked by an graft edge. Since grafts are copies of subtrees of the original contract, execution can never diverge from the intended behaviour.

Off-chain behavior includes the on-chain one When all participants are honest, they are able to emulate any execution step of the on-chain contract by signing the corresponding graft. The required unlocking conditions are the same.

In the presence of adversary, instead, it is possible that the graft can not be created even when it corresponds to an on-chain step, since the adversary can simply refuse to sign it. An honest participant, however, detects that the adversary is not cooperating with time Δ_r, and uses the failsafe mechanism: Init is put on the blockchain, as well as the root of the most recent graft. Since now the execution is moved on-chain, the signatures from the adversary are no longer required. Indeed, all the implicit signatures have already been provided during the graft creation. The honest participant can therefore execute the wanted step.

Note that when executing a contract on-chain it is possible that multiple (possibly honest) participants attempt to perform distinct execution steps, causing a race on the blockchain between multiple transactions that spend the same output. When executing the same contract off-chain, participants have several options to handle such races. Ideally, participants should attempt to reach an off-chain agreement on what should be the next step to take (as per Item 1 of Definition 30). For instance, participants could run a sub-protocol to securely generate a random number and use it to choose among the multiple options. If an agreement can not be reached on the next step, the execution must be moved on-chain. Doing so causes a race on the blockchain, exactly as in the on-chain execution.

Finality of the execution A more subtle adversary could attempt to disrupt the execution not by making it diverge from the contract or by stalling it, but by rolling it back to a previous state.

Recall that, after off-chain stipulation, Head has been appended to the blockchain, while Init has been signed by all the participants and can be used to redeem Init at any time. Further, the off-chain execution protocol repeatedly sign grafts that can redeem Init: one graft for each off-chain execution step. During this, the adversary receives a sequence of signed grafts.

With this knowledge, the adversary can try to append Init and then redeem that using the root of *any old graft*, effectively moving the execution of the contract on-chain while restarting it from an *old* state. This would be problematic in several ways. First, the adversary could make the contract progress on-chain following contract tree path which is different from the one that has already been followed off-chain, which is undesirable. Second, when executing this different path, the adversary has still access to the secrets that have been revealed during the off-chain execution of the other path. This

might be harmful to the participants which would not have wanted to reveal such secrets in this different path.

To avoid such a roll-back attack, our off-chain execution protocol exploits temporal constraints on grafts. More precisely, grafts are created so that they can only redeem Init after a certain time has passed. Newer grafts are given a shorter time lock, so that they become enabled to redeem Init *before* all the previously created grafts. Effectively, newer grafts have higher priority. To thwart roll-back attacks, honest participants must constantly be vigilant and detect when someone appends Init on the blockchain (Item 2 of Definition 29). After that happens, honest participants must append the root of the *latest* graft as soon as it becomes enabled. Since this spends Init, it prevents older grafts to be used. Finally, the priority mechanism ensures that this counter-attack always succeeds.

Denial of Service attacks Protocols which make use of timelocks are susceptible to Denial of Service attacks if the adversary is able to delay the actions of participants. This threat applies both to the on-chain and off-chain executions. Still, several Bitcoin timelock-based protocols are frequently used in practice (e.g., HTLC [6], micropayment channels [94]). These protocols are considered to be secure since it is unrealistic for an adversary to be able to completely cut participants off the Bitcoin network for a very long time. Using our notation, as long as Δ_w is chosen to be large enough, we can realistically assume that participants are always able to react in time when needed.

Our off-chain execution protocol also uses a shorter time Δ_r to detect unresponsive participants. Depending on the actual value of Δ_r, a DoS attack that delays a participant for Δ_r or longer could be realistically feasible. Still, we remark that the impact of such attack would not be catastrophic, and can only cause the contract execution to be moved on-chain (and, consequently, fees to be paid). The impact is therefore the same as the one caused by a misbehaving participant that stops cooperating.

7.4 On-chain vs off-chain protocols: comparison and assessment

We compare our protocol for off-chain execution against the standard on-chain contract execution protocol. The main benefits of our new protocol are the following:

- **Only three fees in the best case.** If the contract is brought to completion through off-chain steps, then only three transactions are appended on the blockchain, independently from the size of the original contract. This significantly reduces the cost for deep contracts.

- **No more than two additional fees in any case.** Indeed, the worst case is met when some adversary forces the participants to move the execution on-chain after the stipulation phase, e.g., by redeeming Head with Init immediately. In such case, participants only have to pay for the on-chain execution costs, plus two more fees for Head and Init.

  Note that if the participants complete at least two off-chain steps, that is already enough to bring the execution cost on par with the original on-chain contract, even if some participant misbehaves on the third step. Moreover, any additional off-chain step beyond the second one further reduces the cost of recovering from future adversarial behaviours.

- **No state rollbacks.** After a honest participant completes an off-chain step reaching a contract node, no adversary can put on-chain a transaction corresponding to an earlier node, effectively rolling back one or more execution steps. Indeed, after Init is appended on-chain, the next transaction to be appended must be the root of some graft. The graft corresponding to the last off-chain step has a shorter timelock w.r.t. other graft, so the honest participant is able to put its root on-chain before any adversary can do the same with another graft.

- **No additional waiting in the best case.** With every off-chain step, the timelock on the next graft decreases, reaching 0 when a leaf is reached.

On the other hand, the main drawbacks of off-chain execution are the following:

- **Two additional fees in the worst case.** As already mentioned above, in the worst case we do have to pay two more fees.

- **Redeeming Init can require waiting.** Except in the best case (where the contract is completed off-chain), the failsafe mechanism requires the grafted subtrees to have timelocks. This means that whenever a participant is forced to move the contract execution on-chain due to some misbehaving participant, they have to wait before they can redeem Init with their graft root. This is more harmful in the case of an early attack in the first off-chain steps, since the graft timelocks are larger. Also, moving the execution on-chain early will make the contract consume more fees.

- **Increased number of signatures and messages.** At every off-chain step, the participants need to exchange signatures for every transaction in the grafted subtree. Hence, performing many steps requires participants to exchange more signatures compared to the on-chain stipulation and execution phases. The worst case scenario happens when the original contract tree has the form of a long chain of n nodes, each of them having only one child. Executing this contract off-chain requires to sign $O(n)$ grafts, each of which contains $O(n)$ transactions on average. Overall, this requires each participant to exchange $O(n^2)$ additional signatures.

When considering fees, our protocol provides significant benefits in almost every scenario. The main drawback seems to be due to timelocks, since a malicious participant can delay the completion of a contract by an amount of time that grows linearly with the depth of the contract tree. The increased number of signatures does not seem to be too detrimental, especially when considering that for balanced contract trees the overhead is only of $O(n)$ messages.

Incentivizing honest behaviour We remark that, when the contract leaves redistribute the unspent fees to the participants, all the participants are encouraged to behave honestly. This is because forcing the contract execution to be moved on-chain provides no actual benefit for adversaries, but reduces the amount of unspent fees that will be refunded.

Limitations We remark a few limitations of our approach. First, participants must always be live and monitor the blockchain for malicious behaviours, reacting to them in a timely fashion. Doing so assumes that adversaries are not able to perform DoS attacks that can stall honest participants for a significant amount of time. Indeed, stalling a participant for more than Δ_w can unlock a *wait*: t condition before that participant can react, disrupting the contract execution in both the on-chain and off-chain case. For this, we assume that Δ_w is long enough to ensure that honest participants have plenty of time to act, even in presence of DoS attacks or severe congestion of the Bitcoin network. Note that this assumption is widespread, since it is applied to all the time-based blockchain protocols, such as hashed time locked contracts [12], lotteries [6, 31], and micropayment channels [54, 94]. Indeed, the security of such protocols is grounded on the assumptions that honest participants can not be prevented to append their transactions in a timely fashion.

By contrast, stalling a participant for the shorter time Δ_r is not as disruptive. In the wort case, that participant can be regarded as unresponsive by the others, prompting them to move the contract execution on-chain. While this is not ideal because we now have to pay more fees, at least the contract semantics is preserved. Moreover, if an adversarial participant aims to move the contract on-chain, they can simply stop interacting with others, which is far easier than performing a DoS attack on others.

Our approach computes the timelocks according to the depth of the original contract tree, which must be finite and statically known. This assumption is shared with the on-chain execution protocol. This is inevitable when dealing with the Bitcoin platform, due to the limited expressiveness of its scripting language.

Finally, in our protocol, all the contract fees must be provided in advance, and locked within the Head transaction. This applies to both the on-chain and off-chain protocols. Additionally, we must specify in advance how much each transaction in the tree will pay as fee: this happens at signing time, much sooner than when the fees are actually paid. Only after the contract is terminated the unspent fees are refunded to participants. In the off-chain protocol the problem is partially mitigated by the fact that fees could be adjusted when creating the new grafts, if the current market conditions suggest to do so. However, the fees in the grafts must still be covered by the Head balance, which limits the possible adjustments. Further, several hours may pass between the signing of a graft and its actual use, making it harder to

accurately predict the appropriate fees. Finally, once the execution is moved on-chain fees are again locked.

Protocol improvement: saving one more fee As discussed above, our protocol is efficient in terms of fees, especially in the best case where all the participants are honest and only *three* fees are needed. It is possibly to slightly improve the best case and make it so only *two* fees are needed. For that, it suffices to amend the grafting protocol. Recall that, when the contract execution is about to end and we create the last graft, involving only a single transaction T for a leaf, our protocol signs T so that it can redeem Init. After that, we can redeem Head with Init and then Init with T, closing the contract, having paid a total of three fees. We modify the grafting protocol so that after creating T we also create a T' with the same outputs, but having Head as its input instead of Init. T' is then signed accordingly. In this way, to close the contract we can directly redeem Head with T', which involves paying a total of two fees only.

Protocol improvement: reducing the number of signatures Both the on-chain and off-chain contract stipulation and execution protocols employ a fairly large number of signatures. Indeed, virtually all the transactions require at the very least the implicit signatures, whose number is the same as the number of the participants. This impacts on the script size, which should be kept small. To improve script size, Bitcoin allows the use of Schnorr signatures [91]. These allow to combine the signatures of many (cooperating) participants into a single signature. At stipulation time and at grafting time (when the participant cooperate to sign a new graft) participants can communicate to produce a Schnorr signature for the transactions at hand instead of sharing a set of signatures as usual. Transactions requiring execution-time authorizations from multiple participants can also benefit from Schnorr signatures, again reducing the number of signatures to a single one. Using this technique, we essentially reduce the script of each transaction to be a logical disjunction involving as many cases as the number of children in the contract tree, where each case requires a single Schnorr "implicit" signature, plus another one when authorizations are required by the contract logic. Note that we can not combine these two signatures since one is created at stipulation/grafting time, while the other is created at execution time. This makes the script size no longer dependent on the number of participants.

7.5 Conclusions and related work

We presented an optimistic protocol for the off-chain execution of Bitcoin smart contracts. Our protocol allows participants to save on transaction fees. In the best case scenario, we only require three fees, while in the worst case we need to pay the cost of the original contract, plus two fees.

Our protocol achieves both security and efficiency by leveraging floating transactions, i.e. transactions that are signed off-chain but not yet put on the blockchain. More precisely, the security of our protocol is based on the fact that honest participants hold the latest graft floating, and are able at any time to put the graft on chain in order to commit the latest state to the blockchain. Instead, the efficiency of our protocol follows from participants not having to put these floating transactions on chain until the very end of the contract (if all participants are honest). In general, off-chain protocols (also referred as *layer 2* protocols) are a common solution to improve the scalability of blockchains [63, 113].

Our mechanism of floating transactions is reminiscent of the one exploited by the Lightning Network Protocol [94]. The Lightning Network exploits floating transactions to realize *micropayment channels*, allowing participants to transfer bitcoins by simply signing a new floating transactions, without having to append anything on the blockchain and pay the associated fees. Such payments are also instantaneous, since there is no need to wait for transactions to be confirmed. While such use of floating transactions is a widely adopted and studied approach, so far on stock Bitcoin it has only been used for specific purposes (e.g. micropayment channels). By contrast, our protocol is general, and can be used to run *any* contract tree off-chain.

We are aware of a few other approaches to improve the execution of Bitcoin contracts. For instance, [53] proposes *eltoo*, a "layer 2" protocol to move contracts off-chain. This protocol relies on extending Bitcoin with a special signature type (a special *sighash*), which is exploited to allow jumping from an old state to a newer one without having to perform the intermediate steps. By contrast, our approach is tailored for Bitcoin as-is, without any extensions.

Another approach [51] is to execute contracts off-chain in a controlled way, and then require the result to be certified before it can be put back on the blockchain. This approach is feasible but it requires and external trusted execution environment. While trusting the hardware is an option, this approach effectively gives the hardware manufacturers the status of a

certification authority, and so the power to break contract execution. As such, contract are no longer decentralized. Our approach instead only relies on basic cryptography, time locks, and the Bitcoin network.

Smart contracts can be implemented on top of Bitcoin [86,103] by creating a new "layer 2" overlay network with its own consensus mechanism which commits the contract state on Bitcoin. This is sometimes referred to as the "side chain" approach. Note that, in this setting, the security of the smart contracts no longer relies on the honesty of the majority of the Bitcoin nodes, but also on that of the layer 2 nodes. Our approach does not relay on any external complex infrastructure beyond stock Bitcoin. A well-known approach to efficiently run smart contracts in several platforms is to use *rollups* [111] A first form of rollup is given by *zero-knowledge rollups* [47]. In this approach, contracts are first executed off-chain, and then their result is put on-chain together with a zero-knowledge proof that the result is indeed the intended one. Alternatively, *optimistic rollups* [72] can be used: after the contract is run off-chain, any participant can attempt to put a state on chain and claim it is the correct result. After that, other participants can verify the claim and challenge it if they believe the state to be wrong. Such challenges must be made within a time window, after which the state becomes final. Winning such challenge rejects the proposed new state, and penalizes the proponent. Overall, these rollup protocols appear to be very useful but they are designed for powerful platforms like Ethereum where their complex on-chain mechanisms can be implemented. On stock Bitcoin, instead, we only have a very limited scripting language, so it is unclear if the approach can be followed.

Part III

Contributions: Formal analysis of MEV

7.6 Motivating example: the Automated Market Maker

In this section we informally analyse the MEV of Automated Market Makers (AMMs), in order to provide the intuitions that will guide our Lean formalization in Section 8.5.

In general, a smart contract can be seen as a public API through which users exchange tokens. Users interact with a contract by sending *transactions* to the blockchain network. These transactions are signed by the sending user to ensure integrity and authenticity. The blockchain network collects the pending transactions in a public *mempool*, from which special nodes, called validators, select and sequence the transactions that will form the next block to be appended to the blockchain. The sequence of transactions in the blockchain determines the current state of both users' wallets (i.e., their token holdings) and the states of deployed contracts (including their wallets) — which collectively form the blockchain state. At an abstract level, we can then see a blockchain as a state transition system, where state transitions are triggered by transactions.

Maximal Extractable Value attacks The ability to craft a block and choose which pending transaction to include in it gives the validator's a significant edge over other users. Indeed, a validator can include their own transactions in a block, and, crucially, they can do so with the full knowledge of the rest of the block (since they are the one constructing it). By creating a block that interleaves user's transactions with their own, a validator is able to *extract value* from the users, essentially using the information about the pending transactions to perform an arbitrage. This manipulation of transaction ordering that profits at the expense of regular traders is known as a maximal extractable value attack, or MEV attack. In the following chapter we will focus on MEV attacks, in particular proving the optimality of a certain attack strategy against an Automated Marked Maker (AMM) contract.

Automated Market Makers AMMs are smart contracts that enable users to swap between two token types according to an algorithmically defined exchange rate. For illustration, we present in Listing 7.1 a simplified implementation of the AMM contract. When the contract is first initialized,

Listing 7.1: A constant-product AMM contract in Solidity (simplified).

```
contract AMM {
  uint r0, r1; // reserves of token types t0, t1
  constructor(uint n0, uint n1) {
    t0.transferFrom(msg.sender, address(this), n0);
    t1.transferFrom(msg.sender, address(this), n1);
    r0 = n0; r1 = n1;
  }
  function swap0(uint n0, uint x1_min) {
    t0.transferFrom(msg.sender, address(this), n0);
    uint x1 = (n0 * r1) / (r0 + n0); // compute output tokens
    require (x1>=x1_min && x1<r1);
    t1.transfer(msg.sender, x1);
    r0 = r0 + n0; r1 = r1 - x1;
  }
  function swap1(uint n1, uint x0_min) { /* symmetric to swap0 */ }
}
```

its `constructor` is called providing the reserves of tokens T_0 and T_1. After that, the other methods allow users to swap tokens according to the Uniswap v2 [1] mechanism, which computes exchange rates so to maintain the product of reserves (i.e., `r0` $\cdot$ `r1`) constant. For instance, a swap order doubling the amount of T_0 in the current contract reserves must receive in exchange half the amount of T_1 in the current contract reserves.

Concretely, the method `swap0` allows a trader to pay `n0` units of their T_0 token, and receive units of T_1, where the actual amount is determined by the AMM's internal exchange rate. The parameter `x1_min` is chosen by the trader as a safeguard against the case where such exchange is less advantageous than the expected one. More precisely, if the received amount of tokens T_1 does not reach at least `x1_min`, then the transaction is *reverted*, meaning that no state update happens — including the payment of tokens T_0 from the trader. Note that in blockchains following the account-based paradigm [17], such as Ethereum, a user sending a transaction has no certainty about the state where it will be executed. Therefore, such kind of safeguards are often necessary to avoid unintended deals. The function `swap1` is symmetric: it takes T_1 as input and outputs T_0.

MEV attacks on AMMs AMMs are subject to different forms of MEV attacks, which exploit differences in prices between the internal AMM exchange rate, i.e. the ratio between the AMM reserves, and external market prices, which we model below with the function $\texttt{price}(\mathrm{T})$.

When internal and external prices are disaligned — i.e., if $\texttt{r0} \cdot \texttt{price}(\mathrm{T}_0) \neq \texttt{r1} \cdot \texttt{price}(\mathrm{T}_1)$ — a simple MEV attack is to perform *arbitrage*, i.e. a swap that realigns the internal and external prices. For example, assume that $\texttt{price}(\mathrm{T}_0) = 4$, $\texttt{price}(\mathrm{T}_1) = 9$, and let S be a blockchain state where the AMM reserves are of 6 units of each token, and an adversary Adv owns n_0 tokens of type T_0 and n_1 tokens of type T_1. Formally, we write:

$$S = \texttt{AMM}[6{:}\,\mathrm{T}_0, 6{:}\,\mathrm{T}_1] \mid \mathsf{Adv}[n_0{:}\,\mathrm{T}_0, n_1{:}\,\mathrm{T}_1] \mid \cdots$$

where the $\cdots$ denote parts of the state that are irrelevant for the attack. In this arbitrage attack, the adversary executes a transaction $\mathsf{Adv}{:}\,\texttt{swap0}(3, 0)$, which updates the state to:

$$\texttt{AMM}[9{:}\,\mathrm{T}_0, 4{:}\,\mathrm{T}_1] \mid \mathsf{Adv}[n_0 - 3{:}\,\mathrm{T}_0, n_1 + 2{:}\,\mathrm{T}_1] \mid \cdots$$

The resulting AMM is balanced, and the adversary has paid 3 units of T_0 (with value $3 \cdot 4 = 12$) to buy 2 units of T_1 (with value $2 \cdot 9 = 18$). So, overall Adv has gained $18 - 12 = 6$. Such arbitrage strategy achieves optimal MEV when the mempool does not contain swap transactions signed by traders. We establish this fact formally in Section 8.5.

More sophisticated MEV strategies arise when the adversary can exploit pending transactions in the mempool. In this scenario, the adversary can interleave users' transactions with adversarial ones, possibly obtaining a profit even when internal and external prices are aligned. A typical strategy of this kind is the *sandwich attack* [121], which we illustrate in the same state S as before. Assume that a trader A has sent a transaction $\mathsf{A}{:}\,\texttt{swap0}(3, 1)$ to the mempool. Here, A has set the parameter `x1_min` to 1, requiring a lower bound on the number of T_1 units received from the swap. To perform the sandwich attack, the adversary constructs the following transaction bundle:

1. $\mathsf{Adv}{:}\,\texttt{swap0}(3, 0)$, an arbitrage transaction performed by the adversary;
2. $\mathsf{A}{:}\,\texttt{swap0}(3, 1)$, the transaction picked from the mempool. Since A's transaction is executed in a state where the AMM is in equilibrium (i.e., prices are aligned), A will not receive the 2 units they would have

expected in S: rather, they receive the minimum of T_1 units admitted by the constraint `x1_min`, i.e. only 1 unit. This causes A to have a negative gain, i.e. $1 \cdot \texttt{price}(\mathrm{T}_1) - 3 \cdot \texttt{price}(\mathrm{T}_0) = 9 - 3 \cdot 4 = -3$;

3. $\mathsf{Adv}\colon \texttt{swap1}(1, 3)$. The adversary closes the sandwich with another arbitrage transaction, after which the AMM reaches again the equilibrium.

Overall, executing this transaction bundle in S leads to:

$$
\begin{aligned}
&\texttt{AMM}[6\colon \mathrm{T}_0, 6\colon \mathrm{T}_1] \mid \mathsf{Adv}[n_0\colon \mathrm{T}_0, n_1\colon \mathrm{T}_1] \mid \mathsf{A}[m_0\colon \mathrm{T}_0, m_1\colon \mathrm{T}_1] \mid \cdots \\
\xrightarrow{(1)}\ &\texttt{AMM}[9\colon \mathrm{T}_0, 4\colon \mathrm{T}_1] \mid \mathsf{Adv}[n_0 - 3\colon \mathrm{T}_0, n_1 + 2\colon \mathrm{T}_1] \mid \mathsf{A}[m_0\colon \mathrm{T}_0, m_1\colon \mathrm{T}_1] \mid \cdots \\
\xrightarrow{(2)}\ &\texttt{AMM}[12\colon \mathrm{T}_0, 3\colon \mathrm{T}_1] \mid \mathsf{Adv}[n_0 - 3\colon \mathrm{T}_0, n_1 + 2\colon \mathrm{T}_1] \mid \mathsf{A}[m_0 - 3\colon \mathrm{T}_0, m_1 + 1\colon \mathrm{T}_1] \mid \cdots \\
\xrightarrow{(3)}\ &\texttt{AMM}[9\colon \mathrm{T}_0, 4\colon \mathrm{T}_1] \mid \mathsf{Adv}[n_0\colon \mathrm{T}_0, n_1 + 1\colon \mathrm{T}_1] \mid \mathsf{A}[m_0 - 3\colon \mathrm{T}_0, m_1 + 1\colon \mathrm{T}_1] \mid \cdots
\end{aligned}
$$

This gives the adversary a gain of $1 \cdot \texttt{price}(\mathrm{T}_1) = 9$, which is greater than the gain of 6 achieved through the arbitrage attack, and it actually is the (optimal) MEV in S.

In general, the optimal MEV-extracting strategy depends on a multitude of factors: the AMM reserves, the external token prices, the direction of the swap in the mempool (i.e., `swap0` *vs.* `swap1`), the swap amount and lower bound. The resulting combinatorial explosion of cases and subcases makes manual reasoning about MEV extremely error-prone. We tackle this complexity with the help of the Lean proof assistant, establishing in Section 8.5 the optimality of the sandwich MEV strategy in the general case.

Chapter 8

Certifying optimal MEV strategies with Lean

8.1 Introduction

Public blockchains such as Ethereum currently handle billions of dollars in crypto-assets, controlled by smart contracts that implement increasingly complex financial applications. Users send their orders (represented as *transactions*) to the blockchain network, which sequences them into a public ledger that records all the asset exchanges. In most cases, the underlying protocols of these blockchains do not enforce transaction order fairness, instead delegating the sequencing of user transactions to miners or validators. Consequently, an adversarial validator might insert other transactions right before the user orders (*front-running* attacks) as well as after them (*back-running* attacks), affecting the smart contract state in which the user orders are executed. In this setting, a relevant class of attacks is known as Maximal Extractable Value (MEV) attacks, where adversaries manipulate transaction sequencing for profit [50]. Besides negatively affecting the profits of honest users, the overall impact of MEV is detrimental on the affected blockchains, undermining decentralization, transparency, and exacerbating network congestion [96, 112]. Decentralized Finance (DeFi) protocols, in particular, are common targets of MEV attacks, which overall have led to profits for adversaries worth several billions of dollars in the period [73].

MEV can be approached from two different perspectives, depending on whether one plays the role of attacker or defender. For an attacker, the fact that the value extracted is *maximal* is not really relevant. What truly matters is having an efficient algorithm to bundle one's own transactions with those of other users in a way that guarantees profit. For that purpose, the attacker can exploit known heuristics targeting specific contracts [50, 77, 97], or devise adaptive techniques that can potentially extract value from arbitrary contracts [15]. In both cases, the adversary wins if the value extracted exceeds the value paid to mount the attack.

Playing the role of defender is substantially harder: to guarantee that the value extractable from a contract is bounded by a given threshold v, one must ensure that no adversarial strategy — among an *infinite* set of possible strategies — can extract more than v. More abstractly, let $\mathrm{EV}(S, v)$ be a predicate stating that in system state S the extractable value is bounded from above by v. Then, the adversary's task reduces to *falsification*: finding a counterexample to $\mathrm{EV}(S, v)$, by exhibiting a strategy that extracts some $v' > v$. Instead, the defender task amounts to *verification*: constructing a proof that $\mathrm{EV}(S, v)$ indeed holds.

From this perspective, establishing MEV requires the same fundamental ingredients as program verification, namely: (1) a formal model of the system under analysis, i.e., smart contracts executed on blockchains; (2) a precise formalization of the property of interest; (3) a proof technique to establish the property.

Regarding item (1), the literature proposes several formal models of contracts at different levels of abstraction — ranging from the low-level Ethereum Virtual Machine [44, 62, 66] to the high-level contract language Solidity [49, 69, 81]. By contrast, regarding item (2) most existing definitions [14, 82, 101] lack the precision needed to establish MEV, e.g. they often rely on informally-defined notions, or omit key aspects of adversarial capabilities such as eavesdropping the *mempool*, i.e. the set of pending transactions sent by users. We emphasize that, from a defender's perspective, the accuracy of a MEV formalization is crucial to ensure that no class of attacks is overlooked. Regarding item (3), as noted above, establishing precise MEV bounds requires considering *all* sequences of transactions that an adversary can construct using the knowledge of the transaction mempool. Although in many cases this infinite set of adversarial strategies can be partitioned into a finite number of equivalence classes, the resulting combinatorial explosion of cases and subcases quickly becomes overwhelming, well beyond the reach of reliable pen-and-paper proofs. This explosion occurs even for relatively simple contracts, where just a handful of system variables already gives rise to a vast and intricate strategy space. This is the case, e.g., of Automated Market Makers, a foundational DeFi contract that, despite implementing a simple functionality (the swap of two tokens at an algorithmically-determined exchange rate [7, 8, 115]), gives rise to a complex economic mechanism, leading to subtle attack strategies [20, 121].

To address these challenges, it is necessary to devise a MEV formalization that can capture all possible adversarial strategies, and at the same time is amenable to rigorous, machine-verified proofs that go beyond manual analysis. Proof assistants provide a natural framework to meet these requirements, as they enable precise formalizations of software systems and support the construction of machine-verified proofs whose correctness is guaranteed beyond any reasonable doubt. In the context of MEV, they offer the potential to reason about adversarial strategies, automate parts of the verification process, and ensure a level of rigour that manual analysis cannot achieve. However, despite its central role in the security of decentralized applications, no mechanized formalization of MEV has been developed so far. Existing studies

rely on informal arguments, which are insufficient to provide guarantees of correctness. This leaves a critical gap between the theoretical understanding of MEV and the practical need for security of decentralized applications.

Contributions In this chapter we present the following key contributions:

1. we provide the first fully mechanised formalisation of MEV within a proof assistant. To this purpose, we adopt Lean 4 [84], an open-source theorem prover and programming language that has been successfully applied to construct and verify large-scale proofs [110]. Its extensive mathematical library makes Lean 4 particularly well suited for reasoning about DeFi contracts, which often involve complex mathematical manipulations.

2. we devise a new proof technique for establishing MEV. Roughly, given a system state S where MEV is to be estimated, our proof technique requires the defender to provide a "guess" function mapping each state that can be reached from S to a candidate MEV amount. The defender must then prove that (i) the guess for the initial state is an under-approximation of MEV, and (ii) any adversarial move affects the guess by an amount which is bounded by the gain of the move. We establish that our proof technique is *sound* — i.e., when such a guess function exists, it actually gives the MEV — and *complete* — i.e., when the MEV exists, a guess function exists.

3. we apply our proof technique to the analysis of Automated Market Makers (AMMs). Although the MEV of AMMs has been studied before [19, 75, 118, 121], our work provides the first mechanized proofs of the optimality of *sandwich attacks*, showing that when the attack is applicable, the adversary can extract the maximal possible value.

Our Lean formalization, including additional case studies besides the AMM, is available online in a public repository [2] consisting of ∼7000 lines of code.

8.2 System model

To formalize a generic smart contract, we use the following abstract type parameters:

- `Token`, representing the possible token types exchanged within the system;
- `State`, representing the state of a smart contract running in an honest environment, and intuitively comprises the contract variables, its wallet, the wallets of the honest participants, and the mempool;
- `Move`, representing the possible adversarial moves, e.g., crafting and executing a transaction, or executing a transaction sent from an honest participant to the mempool.

These type parameters will then need to be instantiated in order to formalise specific smart contracts (e.g., as done for the AMM contract in Section 8.5).

Wallets are containers of tokens, possibly of different types. Formally, we model wallets as functions from token types to real-valued amounts [1]

We explicitly separate the adversary from honest users, reflecting the different assumptions about their capabilities. In particular, we assume the adversary to be *wealthy*, i.e., able to spend arbitrarily large amounts of tokens when mounting an attack. Honest participants, instead, own a limited amount of tokens. We formalize these assumptions by defining two distinct wallet types: `Wallet`, holding a non-negative amount of tokens for honest participants and contracts, and `WalletAdv` , holding arbitrary token amounts (possibly negative) for the adversary.

```
def Wallet    (Token : Type) : Type := Token → ℝ≥0
def WalletAdv (Token : Type) : Type := Token → ℝ
```

We model the interactions between the adversary, the honest participants, and the contract as a state transition system. Its states are types `SysState`, which simply extend the contract states with the adversary wallet (Δ):

```
structure SysState {Token State : Type} where
  Δ : WalletAdv Token
  s : State
```

[1] We discuss our choice of working with reals instead of integers in Section 8.6.

Finally, we model the transition system as the structure `System` in Listing 8.1. A `System` provides the concrete `Token`, `State`, and `Move` types for the smart contract at hand. The `semantics` field specifies the behaviour of adversarial moves. Intuitively, this comprises appending honest participants' transactions from the mempool, as well as appending transactions crafted by the adversary. The `semantics` field is a *partial* function, mapping a `SysState` and a `Move` to a new `SysState`. The semantics is undefined (`none`) when it is impossible to perform the given move. In practice, this corresponds to the case where a transaction reverts.

To model token flows we add a few additional fields. A `System` comprises a function `honTokens` that associates to each `State` the cumulative amount of tokens owned by the contract and honest participants (hereafter, referred to as the "honest tokens"). We then postulate that tokens cannot be destroyed or created by transactions: property `preserveTokens` stipulates that a transaction (i.e., the `semantics`) cannot change the overall amount of tokens, counting both the honest tokens as well as the ones owned by the adversary ($\sigma.\Delta$).

Further, we let `System` associate each token type with its *price*: the `tokenValue` field maps any wallet to a real number, denoting the cumulative price of all the tokens in the wallet. We require this value function to be non-negative (on positive wallets) and additive.

Listing 8.2 states a few basic properties on an arbitrary `sys : System`. The value of the empty wallet is zero; the value function preserves subtraction and is monotonic on wallets. These properties follow directly from the system requirements on `tokenValue`.

Listing 8.1: `System` model in Lean.

```
structure System where
  Token : Type
  State : Type
  Move  : Type
  semantics : @SysState Token State → Move → Option (@SysState Token
    State)
  honTokens : State → Wallet Token
  preserveTokens : ∀ σ m, match semantics σ m with
    | .none    => True
    | .some σ' => ∀ τ, honTokens σ.s τ + σ.Δ τ = honTokens σ'.s τ + σ
    '.Δ τ
  tokenValue : WalletAdv Token → ℝ
  tokenValue_nonneg : ∀ f, f ≥ 0 → tokenValue f ≥ 0
  tokenValue_additive : ∀ f g, tokenValue (f + g) = tokenValue f +
    tokenValue g
```

Listing 8.2: Basic properties of systems.

```
theorem tokenValue_zero : sys.tokenValue 0 = 0
theorem tokenValue_sub (f g : WalletAdv _) :
  sys.tokenValue (f - g) = sys.tokenValue f - sys.tokenValue g
theorem tokenValue_monotonic (f g : WalletAdv _) :
  f ≤ g → sys.tokenValue f ≤ sys.tokenValue g
abbrev System.sysState : Type := @SysState sys.Token sys.State
```

8.3 MEV

In this section we present our Lean formalization of MEV and a general proof technique for certifying the MEV of contracts. We start by defining the *gain* of the adversary upon performing moves, and — building upon this — we define MEV. We then introduce our proof technique in the form of a MEV characterization theorem.

Gain We define the adversarial gain between two system states as the difference of the adversarial wallets. A simple transitivity-like property follows.

```
def gainState (σ σ' : sys.sysState) : ℝ := sys.tokenValue σ'.Δ -
    sys.tokenValue σ.Δ

theorem gainState_trans (σ σ' σ'' : sys.sysState) :
  gainState sys σ σ' + gainState sys σ' σ'' = gainState sys σ σ''
```

In order to define the gain achieved by the adversary upon performing a *list* of moves, we first generalize the `semantics` of single adversarial moves. The effect of a list of moves is the composition of the individual effect of each move in the list, discarding those that fail. Hereafter, we will omit long definitions, referring to [2] for their Lean code.

```
def semMoves (σ : sys.sysState) (tr : List sys.Move) : sys.sysState
```

The adversarial gain of a list of moves is given by comparing the initial and final states.

```
def gainMoves (σ : sys.sysState) (tr : List sys.Move) : ℝ :=
  gainState sys σ (semMoves sys σ tr)
```

We establish a few basic properties of `gainMoves`: the empty list gives zero gain, and the gain of the first move in a list can be separated from the gain of the rest of the moves.

```
theorem gainMoves_empty (σ : sys.sysState) : gainMoves sys s [] = 0

theorem gainMoves_step (σ : sys.sysState) (m : sys.Move) (ms : List
    sys.Move) :
    gainMoves sys σ (m :: ms) = match sys.semantics σ m with
                                | .none    => gainMoves sys σ ms
                                | .some σ' => gainState sys σ σ' +
    gainMoves sys σ' ms
```

The gain of a list of moves is bounded above by the value of all the circulating honest tokens. This will be exploited to obtain an upper bound to the extractable value.

```
theorem gainMoves_bound (σ : sys.sysState) (tr : List sys.Move) :
  gainMoves sys σ tr ≤ sys.tokenValue (sys.honTokens σ.s)
```

MEV definition We now formalize the maximal extractable value (MEV). A system state σ has MEV equal to `v` when two conditions are met: (*i*) there is a list of moves `tr` that, when performed in σ, gives the adversary a gain of `v`; (*ii*) any list of moves `tr`, when performed in σ, gives the adversary a gain of at most `v`:

```
structure MEV (σ : sys.sysState) (v : ℝ) : Prop where
  trace_reaches_v :    ∃ tr, gainMoves sys σ tr = v
  other_traces_worse : ∀ tr, gainMoves sys σ tr ≤ v
```

We remark that MEV may fail to exist. For instance, if the adversary can extract any real value < 1, but they cannot extract 1, there is no *maximum* value that can be extracted, but only a least upper bound. To account for this, we also define the "least upper bound of the extractable values" `MEVsup`. The definition only requires traces whose gain is arbitrarily close to `v`. We prove that `MEVsup` always exists, is unique, and non-negative. Further, when `MEV` exists, it coincides with `MEVsup`, so `MEV` is unique and non-negative.

```
structure MEVsup (σ : sys.sysState) (v : ℝ) : Prop where
  traces_approx :      ∀ (ε : ℝ+), ∃ tr, gainMoves sys σ tr + ε ≥ v
  other_traces_worse : ∀ tr, gainMoves sys σ tr ≤ v

theorem MEVsup.exists_unique (σ : sys.sysState) : ∃! v, MEVsup sys σ v
theorem MEVsup.nonneg (σ : SysState) (v : ℝ): MEVsup sys σ v → v ≥ 0
theorem MEV.equalMEVsup (σ : sys.sysState) {v : ℝ} : MEV sys σ v →
    MEVsup sys σ v
```

Proving MEV In principle, one could attempt to prove that a system `sys` in state σ has MEV equal to `v` by directly applying the definition of `MEV`. Doing so would require to (i) find a trace giving Adv a gain `v`, and (ii) show that no other trace can extract more. The second task, in particular, might be quite hard, as it requires reasoning on *all* the (infinitely many) adversarial

traces. To address this challenge, we introduce a *MEV characterization theorem* that provides a principled proof technique for establishing that a given value indeed coincides with the MEV. Using our `MEV.characterization` theorem (Listing 8.3) requires following these steps:

1. Fix the system state σ from which the adversary is going to extract MEV.

2. Specify an ***invariant*** `inv` on system states. The invariant must be accompanied with a proof ensuring that it holds on σ (`invariant_init`), and that it is preserved by any adversarial move (`invariant_sound`).

3. Specify a ***guess function*** `MEV_guess` that maps any state satisfying the invariant to a value in $\mathbb{R}_{\geq 0}$, which is a candidate MEV for that state.

4. Find an adversarial trace that extracts exactly `MEV_guess` σ from σ. This ensures that, on the system state σ, the guess function provides a lower bound to the MEV. We call this property ***coherence*** (`MEV_guess_coherent`).

5. Prove that the guess function is an upper bound for MEV. More precisely, we must prove that, if σ_0 is any (invariant-abiding) system state, then moving to another state σ_1 cannot induce a gain for the adversary that is greater than the difference between the guesses. We call this property ***soundness*** (`MEV_guess_sound`):

$$\texttt{MEV\_guess}\ \sigma_0 \geq \texttt{gainState}\ \sigma_0\ \sigma_1 + \texttt{MEV\_guess}\ \sigma_1$$

Once all the steps above are completed, our characterization theorem establishes `MEV` σ `(MEV_guess` σ`)`, proving that the MEV in σ is indeed the one given by the guess function.

The choices of the invariant and of the guess function are *crucial*. Intuitively, the invariant defines an over-approximation of the states that can be reached from σ by the adversary. Choosing a suitable invariant is key to simplify the subsequent steps, since they involve properties that are only required to hold on invariant-abiding states. In our experience, simple invariants are enough to this purpose: e.g., when dealing with the AMM contract it is enough to impose a bound on the size of the mempool. The other crucial step is choosing the guess function. Roughly, this corresponds to estimating

Listing 8.3: MEV characterization soundness.

```
theorem MEV.characterization
(σ : sys.sysState)
(inv : sys.sysState → Prop)
(MEV_guess : (σ : sys.sysState) → inv σ → ℝ≥0)
(invariant_init : inv σ)
(invariant_sound : ∀ {m : sys.Move} {σ₀ σ₁ : sys.sysState},
sys.semantics σ₀ m = some σ₁ → inv σ₀ → inv σ₁)
(MEV_guess_coherent : ∃ tr : List sys.Move,
gainMoves sys σ tr = MEV_guess σ invariant_init)
(MEV_guess_sound : ∀ {m : sys.Move} {σ₀ σ₁ : sys.sysState},
(hmove : sys.semantics σ₀ m = some σ₁) →
(σ₀_inv : inv σ₀) →
let σ₁_inv :=  invariant_sound hmove σ₀_inv
gainState sys σ₀ σ₁+ MEV_guess σ₁ σ₁_inv
≤ MEV_guess σ₀ σ₀_inv) :
MEV sys σ (MEV_guess σ invariant_init)
```

the value extracted by an adversary following the optimal strategy. Fortunately, this estimate must only be provided for the states satisfying the invariant, simplifying this task.

Our Lean formalization of the MEV characterization theorem closely follows the previous informal discussion, with the only technical difference that one more argument must be passed to the guess function to ensure the invariant holds.

Below, we provide an informal sketch for the Lean proof of our characterization theorem.

Proof (sketch). We must prove that `MEV_guess` σ is the MEV in σ. The item `traces_reaches_v` follows directly from `MEV_guess_coherent`. For `other_traces_worse`, consider an arbitrary trace $m_0, \ldots, m_{n-1}$ of moves starting from $\sigma_0 = \sigma$ and going through states $\sigma_1, \sigma_2, \ldots, \sigma_n$. We have the following

chain of inequalities:

$$
\begin{aligned}
& \texttt{gainMoves}\ \sigma\ [m_0, \ldots, m_{n-1}] \\
= \ & \texttt{gainState}\ \sigma\ \sigma_1 + \cdots + \texttt{gainState}\ \sigma_{n-1}\ \sigma_n \\
\leq \ & \texttt{gainState}\ \sigma\ \sigma_1 + \cdots + \texttt{gainState}\ \sigma_{n-1}\ \sigma_n \quad + \boxed{\texttt{MEV\_guess}\ \sigma_n} \\
\leq \ & \texttt{gainState}\ \sigma\ \sigma_1 + \cdots + \texttt{gainState}\ \sigma_{n-2}\ \sigma_{n-1} + \boxed{\texttt{MEV\_guess}\ \sigma_{n-1}} \\
\leq \ & \texttt{gainState}\ \sigma\ \sigma_1 + \cdots + \texttt{gainState}\ \sigma_{n-3}\ \sigma_{n-2} + \boxed{\texttt{MEV\_guess}\ \sigma_{n-2}} \\
\leq \ & \cdots \quad \leq \quad \boxed{\texttt{MEV\_guess}\ \sigma}
\end{aligned}
$$

In the chain of inequalities above we first apply `MEV_guess` $\sigma_n \geq 0$, and then repeatedly apply the `MEV_guess_sound` inequality, replacing at each step the last part of the sum `gainState` $\sigma_i\ \sigma_{i+1} +$ `MEV_guess` σ_{i+1} with its upper bound `MEV_guess` σ_i. At the end, the whole sum is reduced to `MEV_guess` σ. □

Our `MEV.characterization` theorem ensures that if there exists a sound and coherent guess function, then that function actually provides the MEV. We also show that our characterization is *complete*: whenever MEV exists, there also exists a sound and coherent guess function (Listing 8.4).

An informal sketch of our Lean proof follows.

Proof (sketch). Since by assumption MEV exists, we choose as guess function the one that maps each (invariant-abiding) state to its MEV, exploiting the axiom of choice. Coherence is straightforward, since the definition of MEV (`trace_reaches_v`) ensures that there is a trace extracting that value.

For soundness, let `m` be a move from state σ_0 to σ_1. We also let `tr` be the trace providing MEV for σ_1 (`trace_reaches_v`), hence `gainMoves tr = guess` σ_1. The definition of MEV (`other_traces_worse`) ensures that the value extracted by any trace `tr'` from σ_0 is bounded by the MEV. In particular, choose `tr' = m::tr`, obtaining `gainMoves` σ_0 `(m::tr)` $\leq$ `guess` σ_0. From this and theorem `gainMoves_step`, we conclude:

$$
\begin{aligned}
& \texttt{gainState}\ \sigma_0\ \sigma_1\ \texttt{+}\ \texttt{guess}\ \sigma_1 \\
= \ & \texttt{gainState}\ \sigma_0\ \sigma_1\ \texttt{+}\ \texttt{gainMoves tr} \\
= \ & \texttt{gainMoves}\ \sigma_0\ \texttt{(m::tr)} \\
\leq \ & \texttt{guess}\ \sigma_0
\end{aligned}
$$

□

Overall, formalizing our system model and the MEV-related theorems required ∼600 lines of Lean code.

Listing 8.4: MEV characterization completeness.

```
theorem MEV.characterization_complete
(σ : sys.sysState)
(inv : sys.sysState → Prop)
(invariant_init : inv σ)
(invariant_sound : ∀ {m : sys.Move} {σ₀ σ₁ : sys.sysState},
sys.semantics σ₀ m = some σ₁ → inv σ₀ → inv σ₁)
(mev : ∀ σ', inv σ' → ∃ v, MEV sys σ' v) :
∃ guess : (σ' : sys.sysState) → inv σ' → ℝ≥0,
-- guess is coherent
(∃ tr : List sys.Move,
gainMoves sys σ tr = guess σ invariant_init) ∧
-- guess is sound
(∀ {m : sys.Move}  {σ₀ σ₁ : sys.sysState},
(hmove : sys.semantics σ₀ m = some σ₁) →
(σ₀_inv : inv σ₀) →
let σ₁_inv := invariant_sound hmove σ₀_inv
gainState sys σ₀ σ₁ + guess σ₁ σ₁_inv ≤ guess σ₀ σ₀_inv)
```

```
contract CoinPusher {
  function push() public payable {
    if (address(this).balance >= 100) {
      address payable rcv = payable(msg.sender);
      rcv.transfer(address(this).balance);
    }
  }
}
```

Figure 8.1: A CoinPusher contract in Solidity.

8.4 The CoinPusher contract

We now consider a case where extracting MEV requires the adversary to leverage transactions pending in the mempool. The *CoinPusher* contract transfers its entire balance to any user whose deposit causes the balance to exceed 100 tokens (see Figure 8.1). The contract has a single function `push`, which receives from the sender any amount `msg.value` of tokens (this transfer happens implicitly along with the call). If the incoming tokens make the contract balance exceed 100 units, the entire balance (including the incoming tokens) is immediately sent to the caller through the `transfer` command.

Let S be a system state where the contract holds 0 tokens and a user A holds 1 token. We assume the adversary Adv to be *wealthy*, i.e. endowed with enough tokens to mount any feasible attack (in practice, Adv can also obtain such tokens as a very short-term loan, even if that might cost of some additional fees). If the mempool in S is empty, then Adv cannot extract any value, hence $\mathrm{MEV}(S) = 0$. Now suppose that the mempool contains a transaction sent by A that calls `push` while transferring 1 token — written A:`push(value=1)`. In this case, Adv can atomically execute the transaction bundle:

A:`push(value = 1)` Adv:`push(value = 99)`

which yields a gain of 1 to Adv. Since no strategy achieves a higher gain, we conclude that $\mathrm{MEV}(S) = 1$. Devising an optimal strategy for an arbitrary contract when the mempool contains many transactions is hard, also due to the combinatorial explosion of possible interleavings. In the CoinPusher case, the adversary's optimal strategy is to include the pending mempool transactions while interleaving its own `push` calls. In the idealized case where there are no transaction fees, this can yield a gain equal to the contract

balance in S plus the total value of those mempool transactions with value < 100.

MEV of the CoinPusher contract

We now formalize the CoinPusher contract we described in §8.4. Some parts are similar to the Airdrop contract: e.g., types `Participant` and `Token` are the same, since we use the same participants and token types. Transactions are instead modelled by the new type:

```
inductive Tx
| push (P : Participant) (v : ℝ+) : Tx
```

A transaction `push P v` sends `v` tokens (of type τ_0) to the contract from participant `P`, causing `P` to win the entire contract balance if `v` tips such balance over the threshold.

The `State` type is a structure that denotes the state of the blockchain without the adversary. More specifically, the field `bal` represents the contract's balance, while the field `wal` denotes the aggregated wallets of all honest participants. Since MEV analysis does not require distinguishing between individual honest users, their holdings are abstracted into this single cumulative wallet. The field `mempool` represents the transactions previously broadcast by honest participants but not yet finalised on-chain. Formally, we represent the mempool as an association list mapping transaction identifiers (`TxId`, which is just an alias for natural numbers) with their associated transactions. Finally, the state has a `threshold` field that keeps track of the value that triggers the payout in the CoinPusher.

```
structure State where
  bal : Token → ℝ≥0
  wal : @Wallet Token
  mempool : AssocList TxId Tx
  threshold : ℝ+
```

We denote the associated system state as `CPState`.

```
abbrev CPState := @SysState Token State
```

We now turn to defining the type of adversarial moves. To this aim, we start by identifying the subset of transactions that can be crafted by the adversary alone. In our scenario, the adversary can only craft drop transactions signed by `Adv` itself. This is formalized by property `advTx`.

```
inductive advTx : Tx → Prop
| push {x} : advTx (.push .Adv x)
```

The `Move` type captures all possible adversarial actions, namely: (i) craft a transaction using its own knowledge (i.e., `Adv`'s private key) and append it directly to the blockchain (`adv`), or (ii) select a transaction from the mempool and include it in the blockchain (`mempool`).

```
inductive Move
| adv : (t : Tx) → advTx t → Move
| mempool : TxId → Move
```

The semantics of a `Move` is to cause an `CPState` update, or fail, coherently with `System.semantics`. More precisely, an `adv` move consists of a `push` transaction on behalf of an adversary, while a `mempool` move looks up the corresponding transaction in the `State.mempool` field and performs it. If a mempool transaction succeeds, it is removed from the `mempool` field so that it is not reused later on.

```
def semTx (tx : Tx) (σ : CPState) : Option CPState := ...
def semMove (σ : CPState) (m : Move) : Option CPState :=...
```

With the semantics of moves, we can finally define the CoinPusher contract

```
def CoinPusher: System := ...
```

We now turn to the study of MEV. We study the MEV of the CoinPusher in two distinct cases:

- the case where the mempool is empty, and
- the case where the mempool contains one transaction.

We do not formalize the general case with $N > 1$ transactions in the mempool. Note, however, that it is a simple extension of the second case: roughly, the strategy of the adversary is to apply iteratively N times the strategy for the singleton mempool.

In the case of an empty mempool, we declare the following invariant.

```
def CoinPusher_empty_invariant (σ : CPState) : Prop :=
  σ.s.mempool.isEmpty
```

If there are no transactions in the mempool, the optimal strategy for the adversary is to `push` enough tokens to trigger the win. For the sake of simplicity, below we choose to send σ`.s.threshold` tokens, even if a smaller amount (σ`.s.threshold - `σ`.s.bal .`τ_0) would also suffice.

```
def CoinPusher_strategy_empty (σ : CPState) :
    List Move := [.adv (.push .Adv σ.s.threshold) .push]
```

We define a guess function for the MEV, posing that we can extract the entire contract balance as MEV.

```
def CoinPusher_empty_MEV_guess (σ : CPState)
    (σ_inv : CoinPusher_empty_invariant σ) : ℝ≥0 :=
  σ.s.bal .τ_0
```

We can prove that our guess function is sound and coherent (exploiting our strategy). This makes it possible to invoke our MEV characterization theorem and establish MEV for the empty mempool case.

```
theorem MEV_CoinPusher_empty (σ : CPState) :
  σ.s.mempool = .nil →
  MEV CoinPusher σ (CoinPusher_empty_MEV_guess σ)
```

We now turn to the case where the mempool contains one transaction, i.e., it is a singleton list. Note that we have to account for the mempool becoming empty after its transaction is appended, so the invariant actually requires that the mempool is either a singleton or empty. For the sake of simplicity, we also require that if the mempool is a singleton, its transaction is from the honest participants. Indeed, adversarial transactions in the mempool are irrelevant for the purposes of MEV, so we can ignore them. The invariant the following:

```
def CP_one_or_less_invariant (σ : CPState) : Prop :=
 σ.s.mempool.isEmpty ∨
 (∃ idx tx, σ.s.mempool = .cons idx tx .nil ∧ tx.P ≠ .Adv)
```

The guess function is more complex w.r.t. the empty mempool case. We define it by cases, according to the mempool:

1. If the mempool is empty, our guess is σ`.s.bal .`τ_0, the same we mentioned earlier.

2. Otherwise, if the mempool is the singleton `tx` (a `push` transaction from the honest participants), the best strategy for the adversary is (i) first,

trigger a win in the contract, emptying its balance, (*ii*) attempt to append `tx`, (*iii*) trigger the win again. The first step extracts σ`.s.bal .`τ_0 tokens, emptying the contract balance.

The second step, appending `tx`, might fail if the honest participant does not have enough tokens to execute it, i.e. if `tx.v >` σ`.s.wal .`τ_0, in which case it has no effect. Further, even if `tx` succeeds, it might send to the contract enough tokens to immediately trigger the win for the honest participant, if `tx.v` $\geq$ σ`.s.threshold`. In this case, the tokens are immediately sent back to the participant, and the transaction has no net effect, since the contract balance was empty. Otherwise, `tx` succeeds and sends fewer tokens than the threshold, loading the contract balance with `tx.v` tokens. Overall, while the second step does not make the adversary directly gain anything, it can potentially load the contract balance.

Finally, the third step extracts the new contract balance. This could be `tx.v` or zero depending on whether `tx` succeeded in loading the contract or not, respectively.

Coherently with our strategy, we guess the MEV to be σ`.s.bal .`τ_0 `+ tx.v` or just σ`.s.bal .`τ_0.

```
def CoinPusher_one_or_less_MEV_guess (σ : CPState)
    (σ_inv : CP_one_or_less_invariant σ) : ℝ≥0 :=
  match σ.s.mempool with
  | .nil => σ.s.bal .τ₀
  | .cons id tx .nil =>
    if tx.v < σ.s.threshold ∧ tx.v ≤ σ.s.wal .τ₀
    then σ.s.bal .τ₀ + tx.v
    else σ.s.bal .τ₀
  | .cons id tx (.cons id2 tx2 rest) => by
    exfalso; ... -- contradicts σ_inv

def CP_strategy_singleton_mempool (σ : CPState)
    (id : TxId) (tx : Tx) : List Move :=
  [ .adv (.push .Adv σ.s.threshold ) .push
  , Move.mempool id
  , .adv (.push .Adv σ.s.threshold ) .push ]
```

We can prove that our guess function is sound and coherent (exploiting our strategy). This makes it possible to invoke our MEV characterization theorem and establish MEV for the singleton mempool case.

```
theorem MEV_CoinPusher_singleton
  (σ : CPState) (id : TxId) (tx : Tx)
  (singleton : σ.s.mempool = .cons id tx .nil)
  (nonadv : tx.part ≠ .Adv) :
  MEV CoinPusher σ
    (if tx.v < σ.s.threshold ∧ tx.v ≤ σ.s.wal .τ₀
     then σ.s.bal .τ₀ + tx.v
     else σ.s.bal .τ₀)
```

We remark that this result extends to the general case where the mempool contains any number of transactions. The optimal strategy here is to attempt to append all the mempool transactions, while interleaving them with an adversarial transaction that triggers the win and empties the contract balance: $[\texttt{trigger}_0, \texttt{tx}_1, \texttt{trigger}_1, \texttt{tx}_2, \ldots, \texttt{tx}_n, \texttt{trigger}_n\ ]$.

Overall, the formalization of the CoinPusher contract and the proofs to establish its MEV amount to ∼1000 lines of Lean code.

8.5 MEV of the AMM contract

In this Section we certify the optimality of arbitrage and sandwich attacks on Automated Market Makers, which we described in Section 7.6. We start by defining a `System` for the AMM contract. As a first step, we define a `Participant` type consisting of one adversary and infinitely many honest participants (associated to string identifiers):

```
inductive Participant
| Hon: String → Participant
| Adv: Participant
```

The AMM handles two token types, which we denote generically by τ_0 and τ_1. Accordingly, we instantiate the `System.Token` field with the following type:

```
inductive Token
| τ₀ : Token
| τ₁ : Token
```

We instantiate the `System.State` field with the following type. Its fields represent the contract balance (`bal`), the tokens of all honest participants as a cumulative wallet (`wal`), and the current `mempool`. The mempool is an association list between transaction IDs (`TxId`, represented as natural numbers) and transactions (to be defined below). The resulting system state, which also includes the adversary wallet, is then denoted with `AMMState`.

```
structure State where
  bal : Token → ℝ+    -- quantity of tokens τ₀ and τ₁ in the contract
  wal : @Wallet Token  -- quantity of tokens τ₀ and τ₁ owned by all
    honest participants
  mempool : AssocList TxId Tx
abbrev AMMState := @SysState Token State
```

The only transactions supported by our AMM are token swaps:

```
inductive Tx
| swap (P : Participant) (v₀ : ℝ+) (τ : Token) (vmin : ℝ≥0) : Tx
```

When a `swap P v₀ τ vmin` is executed, participant `P` exchanges `v₀` of their own τ tokens with at least `vmin` tokens of the other type in the AMM reserves. If such an exchange is impossible, i.e. when `P` does not own enough tokens or when the AMM exchange rate would make `P` receive fewer than `vmin` tokens, the transaction reverts.

We instantiate `System.Move` with the following type. Move `adv` allows the adversary to perform their own swap transaction: the predicate `advTx` ensures that `P` is `Adv`. This move updates the adversary's wallet $\triangle$ and the contract balance `bal`. Move `mempool`, instead, allows the adversary to append a honest participant's transaction from the mempool. This move updates the participant's wallet `wal` and the contract balance `bal` accordingly, and it removes the used transaction from the mempool.

```
inductive Move
| adv: (t: Tx) → advTx t → Move    -- uses a tx crafted by the adversary
| mempool: TxId → Move              -- uses a tx from the mempool
```

We instantiate the field `System.semantics` so to associate each move to its above-mentioned effects. We instantiate the field `System.honTokens` mapping each `State` to the sum of its `bal` and `wal` fields. We then prove `System.preserveTokens`.

Finally, we assume a `pr` function that provides the external price (a positive real) to each token type, and exploit it to instantiate `System.tokenValue` and prove its related properties `tokenValue_nonneg` and `tokenValue_additive`.

```
def Prices : Type := Token → ℝ+
variable (pr: Prices)
```

This completes the definition of `AMM : System`.

Arbitrage In the case where the mempool is empty, we show that the arbitrage attack achieves the MEV, i.e. it is optimal. To this purpose, we exploit our MEV characterization theorem. For the invariant, we use the following, which we easily prove to be sound, i.e. preserved by any `Move`:

```
def empty_mempool_invariant (σ : AMMState) : Prop := σ.s.mempool.isEmpty
```

For the guess function (Listing 8.5), we start by defining `extractable` σ as the value that can be extracted by performing a swap that realigns the internal AMM prices with the external ones, refining Theorem 9 in [19]. We also use this value as our guess in σ:

```
def extractable (σ : AMMState) : ℝ≥0 :=
  ⟨ ( Real.sqrt (pr .τ₀ * σ.s.bal .τ₀) - Real.sqrt (pr .τ₁ * σ.s.bal
    .τ₁) )^2, ... ⟩
def AMM_MEV_guess (σ : AMMState) (σ_inv : empty_mempool_invariant σ) : ℝ
    ≥0 :=
  extractable σ
```

We finally prove that our guess is indeed the MEV. This effectively guarantees that the best adversarial strategy, when the mempool is empty, is to realign the AMM, establishing the optimality of the arbitrage attack.

```
theorem MEV_empty_mempool (σ : AMMState) (σ_inv : σ.s.mempool = .nil) :
  MEV (AMM pr) σ (AMM_MEV_guess pr σ σ_inv)
```

Sandwich attack We now turn to the case where the mempool contains one transaction `tx`. In this case, we show that the sandwich attack achieves the MEV. For this scenario we define our invariant to require that the mempool contains only `tx` *or is empty*, since `tx` is consumed when it is fired. We constrain `tx` as follows: (i) `tx` is owned by an honest participant (`tx.part` $\neq$ `.Adv`), and (ii) the least amount `tx.vmin` of output tokens is strictly positive (`tx.vmin > 0`).

```
def inv_one_or_less (σ : AMMState) : Prop :=
  σ.s.mempool.isEmpty ∨
  (∃ idx tx, σ.s.mempool = .cons idx tx .nil ∧ tx.vmin > 0 ∧ tx.part ≠
    .Adv)
```

Constraint (i) represents the fact that adversarial transactions in the mempool are pointless, since the effect of `Move.mempool` on such a transaction can already be obtained by the adversary using `Move.adv`). Constraint (ii), on the one hand, is coherent with the fact that real-world traders never send unconstrained swap orders. On the other hand, `tx.vmin = 0` would lead to the corner case in which `MEV` does not exist, but only `MEVsup` exists. Indeed, if `tx.vmin = 0`, the adversary would be able to extract any value strictly less than `MEVsup` by making one of the reserves arbitrarily close to zero, and consequently the other reserve arbitrarily close to infinity.

We now define our guess function, which turns out to be significantly more complex than the one for the arbitrage attack. We have to provide a guess for all the states satisfying the invariant, so we must handle mempools having size zero or one. When the mempool is empty, we guess `extractable` σ, as in the arbitrage case. When there is one transaction `tx` in the mempool, we first check if `tx` can be beneficial to the adversary. This happens when both the following conditions are satisfied:

1. `tx` can successfully be executed in some AMM state, i.e., its honest sender actually owns the tokens `tx` is trying to swap (`tx.inputVal` $\leq$ σ`.s.wal tx.inputTok`).

2. `tx` causes its honest sender to transfer to the contract an amount of tokens whose value (`tx.inputVal * pr tx.inputTok`) is larger than the value of the minimum amount of tokens it is requiring in exchange (`tx.vmin * pr (other_tok tx.inputTok)`).

If either of the above conditions does not hold, `tx` is useless for the adversary, who can then disregard it. In this case, our guess is the same as the one for the arbitrage case, hence `extractable` σ.

Instead, when both of the above conditions hold, we construct a sandwich around `tx` as follows. First, the adversary employs its unlimited token reserves and performs a swap `move`, bringing the AMM to a state σ`'` where executing `tx` will transfer to the honest participant exactly `tx.vmin` tokens. Intuitively, from the point of view of the honest participant, σ`'` is the *worst state* where `tx` can still be executed. Dually, σ`'` is also the *best state* for the adversary in which to execute `tx`. We refer to σ`'` as the *tight state* for `tx`. After the `move`, having reached the tight state the adversary indeed appends `tx` to the blockchain. After that, we reach a new state σ`''` where the mempool is empty, so the adversary proceeds as the arbitrage case, closing the sandwich.

We coherently define our guess function in Listing 8.5 as the sum of:

1. the gain of `move`, i.e., `gainMoves (AMM pr)` σ `[move]`;

2. the gain of `tx`, which is zero, since it does not directly transfer tokens to or from the adversary;

3. the gain from the closing arbitrage, i.e., `extractable` σ`''`.

We remark that the gain of the first step might be negative. The adversary can therefore temporarily *lose* value in that step, which makes the overall MEV strategy non-trivial. In our Lean development, we first define our guess function as having codomain $\mathbb{R}$ (as done in `AMM_MEV_guess'`). We then show that its result is always non-negative, allowing its codomain to be restricted to $\mathbb{R}_{\geq 0}$, and obtain `AMM_MEV_guess` as required by our `MEV.characterization` theorem. This actually shows how the temporary loss of value for the adversary at the first step is compensated by the next steps.

Note that, as a corner case, it is possible that the initial state σ is already tight for `tx`. If so, no `move` is needed, and we simply use `extractable` σ as our guess.

Finally, theorem `MEV_singleton_mempool` establishes that the sandwich attack achieves the MEV. To prove it, we leverage our MEV characterization theorem: first, we prove our guess to be coherent, showing that its value can actually be extracted using a trace; second, we prove our guess to be sound, showing that no adversarial move can extract more value.

Listing 8.5: Guess function and MEV of the sandwich attack.

```
def AMM_MEV_guess' (σ : AMMState) (σ_inv : inv_one_or_less σ) : ℝ :=
  match σ.s.mempool with
  | .nil => extractable pr σ
  | .cons id tx .nil =>
    if tx.inputVal ≤ σ.s.wal tx.inputTok ∧
       (tx.inputVal * pr tx.inputTok > tx.vmin * pr (other_tok
    tx.inputTok))
    then ...
      let σ' := tight_state_from_mempool σ id tx ...
      let σ'' := state_after_tight σ id tx ...
      match move_from_to_state σ σ' with
      | .some move => gainMoves (AMM pr) σ [move] + extractable pr σ''
      | .none => extractable pr σ'' -- σ = σ', no move needed
    else
      extractable pr σ
  | _ => ... -- contradicts the invariant

def AMM_MEV_guess (σ : AMMState) (σ_inv : inv_one_or_less σ) : ℝ≥0 :=
  ⟨ AMM_MEV_guess' pr σ σ_inv , ... ⟩

theorem MEV_singleton_mempool (σ : AMMState) (id : TxId) (tx : Tx)
  (vmin_pos : tx.vmin > 0) (singleton : σ.s.mempool = .cons id tx .nil)
  (part_nonAdv : tx.part ≠ .Adv) :
  let σ_inv : inv_one_or_less σ := ...
  MEV (AMM pr) σ (AMM_MEV_guess pr σ σ_inv)
```

8.6 Limitations

In this section we discuss how our design choices affect the practical applicability of the proposed Lean formalization.

Real *vs.* integer arithmetic In our MEV formalization, we represent token balances and their market values as real numbers. In practice, however, blockchain platforms use high-precision integers to encode such amounts. For instance, in Ethereum it is common to employ the integer type `uint256`, since floating point types are not supported. As a consequence, when a financial contract performs its numerical computations — e.g., calculating the exchange rate in a `swap` of an AMM — the result is inevitably affected by a small rounding error.

Although these errors usually have a negligible economic impact on the real-world usage of the contract, it is possible to craft artificial scenarios where the MEV differs substantially depending on whether rounding errors are present or not. For instance, consider a slightly modified `CoinPusher` where the winning threshold is 9991, and a 0.1% fee is subtracted from the value sent to the contract upon each call to `push`. Consider a state where the contract has zero balance and the mempool contains a `push` transaction where a honest participant is sending 10001 tokens. Using real numbers, the `push` would compute the fee as 10.001 and transfer 9990.999 to the contract — so, slightly less than the threshold required to trigger the win. The adversary could extract a MEV of 9990.999 by back-running the mempool transaction with another `push` of 1.0001 or more, triggering the win. Instead, using integers, the `push` would round the fee to `10`, and transfer `9991` token units to the contract balance — i.e. exactly the threshold value. Here, the MEV is zero, since the mempool transaction would immediately trigger the win for the honest user, hence it is useless for the adversary.

Adapting our system model and MEV formalization to use integers instead of reals is easy, but we anticipate that it would significantly complicate the reasoning needed to establish MEV for certain contracts. For instance, in our AMM, performing two consecutive adversarial `swap`s has the same effect as a combined single `swap`, which simplifies the treatment. This is no longer the case when rounding errors are taken into account. On the positive side, using integers would ensure that MEV always exists. This does not hold with real numbers: an `Airdrop` variant which allows to withdraw (`drop`) any amount *smaller* than its balance has no associated MEV. Indeed, any

strategy can be improved by adding one more `drop` to grab a tiny amount of additional tokens.

Proof automation While Lean does provide several tactics to automate the proof of certain kinds of mathematical goals, in our experience we often wished for more powerful arithmetic simplification tactics. For instance, the proofs for our AMM model often make use of inequalities involving square roots. We managed to prove these only through several manual algebraic steps, invoking each time a suitable theorem from the mathematical library. This could change in the future as Lean improves its tactics.

Adversary model Our system model keeps the adversary syntactically distinct from honest users. This choice requires a certain care when encoding smart contracts into our system model, especially for contracts implementing access control mechanisms. In general, the system designer must ensure that the `Move` type and its associated semantics faithfully capture all the possible adversarial actions. Omitting even a single adversarial move could lead to overlooked attacks.

8.7 Related work

Analysis tools for MEV The seminal work [14] was the first to propose a general definition of MEV and a tool to estimate MEV upper bounds. Their verification technique is based on a symbolic semantics that over-approximates the set of execution paths and their reachable states. Based upon this symbolic semantics, they encode the problem of estimating a MEV bound as a reachability problem. Compared to our work, this technique requires less manual effort, since the designer is only required to provide a specification of the contract (in their domain-specific language), and from that point the tool automatically performs the analysis of MEV. A main drawback of the approach is the lack of scalability (as observed in [15]): even for relatively simple contracts, the exponential blowup of the generated paths may lead the tool to exhaust the computational resources. Another difference with our work is that our notion of MEV is *exact*: when `MEV sys` σ v holds, it actually means that v can be extracted, and there is no adversarial strategy that can extract more. The technique in [14] instead *over*-approximates MEV, without guaranteeing that the verified upper bound can actually be extracted.

The work [15] approached the problem of MEV estimation from a different perspective, by proposing a machine-learning technique to *under*-approximate MEV. Their algorithm takes as input a blockchain state and a mempool, and gives as output a sequence of transactions (containing both transactions crafted by the adversary and transactions from the mempool) that can be played by the adversary to extract value. This approach is more scalable than [14], and allows the adversary to fine-tune the computational resources used by the algorithm in order to match the available hardware. Of course this technique does not guarantee that the synthesised sequence of transactions is optimal: instead, our approach pursues the goal of certifying the optimality of the adversarial strategy.

Formalization of the adversary Our definition of MEV keeps the adversary distinct from the other participants — so being similar in spirit to [14], where MEV is parameterized by the player who extracts it. As noted in §8.6, this requires some extra care when modelling the contract behaviour, to avoid the risk of omitting some adversarial actions. Some approaches to avoid fixing the identity of the adversary have been proposed in literature. The work [101] defines an adversary-agnostic MEV by first considering the MEVs that can

be extracted by any individual participant, and then taking the minimum. As noted in [101], this definition has some drawbacks when extracting value requires the adversary to make upfront payments. In such cases, the MEV is under-estimated as zero, since the minimum also covers the zero value that can be extracted by a "poor" adversary. The notion of *universal MEV* proposed in [33] overcomes this problem by defining MEV as a game where honest players try to minimize the damage, while adversaries try to maximize their gain. A token redistribution mechanism ensures that the adversaries have enough wealth to execute their extraction strategy. The token redistribution is not completely equivalent to our wealthiness assumption about adversaries, since the token to be distributed may not be sufficient for the attack. A wealthiness assumption more similar to ours was used in [26], which introduces the notion of "MEV of wealthy adversaries" by taking the maximum MEV over all possible adversarial wallets.

Analysis of the AMM contract Compared to real-world AMM implementations such as Uniswap [1], our Lean formalization introduces a few simplifications, that overall contribute to keeping our proofs manageable. A first simplification is that in Uniswap, when a user executes a `swap`, part of the input tokens are paid to the protocol as a fee. Studying the effect of swap fees on MEV would require a further complication of the adversarial strategies, who would need to minimize the impact of fees on their own swaps. The Lean formalization in [55] studies AMMs with swap fees, but unlike ours, it does not address MEV. The additional burden introduced by swap fees is, however, already visible in [55] in the analysis of arbitrage (i.e., the optimal swap in a zero-player game). We expect swap fees to impact MEV analysis to an even greater extent.

Besides swaps, concrete AMM implementations typically allow users to deposit and withdraw tokens from the AMM. In particular, deposit transactions can increase the profits obtainable from subsequent swaps [20], and can thus be leveraged by an adversary to amplifly MEV. The work [19] introduces an extended sandwich attack that combines deposit and swap transactions from the mempool with adversary-crafted transactions, showing that this may increase the extractable value. Extending our proof to handle mempools containing arbitrary combinations of deposit and swap transactions would, however, cause a substantial explosion in the number of cases in our proof of MEV optimality.

A different Lean formalization of constant-product AMMs is presented in [95]. Compared to our work — which proposes a general framework for analysing MEV of arbitrary smart contracts — the work [95] is specifically targeted on AMMs, and only deals with arbitrage.

Sandwich attacks on AMMs were originally analysed in [121], and later in [19, 75, 89, 114, 118], among the others. Compared to these works, ours provides the first machine-checked proof of optimality of sandwich attacks.

MEV countermeasures Given the relevance of MEV on the blockchain ecosystem, several techniques to mitigate its effects have been proposed in the past few years. Some of these techniques are applicable to arbitrary smart contracts [16, 34, 40, 42, 65], while some others are specific to certain classes of contracts, such as AMMs [35, 48]. Formally verifying within our Lean framework that some of these techniques achieves the expected MEV reduction would be a challenging extension of our work.

Bibliography

[1] Uniswap token pair implementation, 2021. https://github.com/Uniswap/uniswap-v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol.

[2] MEV Lean formalization, 2025.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

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

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, pages 30:1–30:15, 2018.

[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014. First appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784.

[7] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant Function Market Makers. In *ACM Conference on Advances in Financial Technologies (AFT)*, pages 80–91. ACM, 2020.

[8] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of Uniswap markets. *Cryptoeconomic Systems*, 1(1), 2021.

[9] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 83–92, 2019.

[10] APTOS Foundation. Network - blockchain - execution. https://aptos.dev/en/network/blockchain/execution, 2024.

[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST)*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[12] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. SoK: unraveling Bitcoin smart contracts. In *POST*, volume 10804 of *LNCS*, pages 217–242. Springer, 2018.

[13] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security*, volume 10957 of *LNCS*. Springer, 2018.

[14] K. Babel, P. Daian, M. Kelkar, and A. Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE Symposium on Security and Privacy*, pages 622–639. IEEE Computer Society, 2023.

[15] Kushal Babel, Mojan Javaheripi, Yan Ji, Mahimna Kelkar, Farinaz Koushanfar, and Ari Juels. Lanturn: Measuring economic security of smart contracts through adaptive learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1212–1226. ACM, 2023.

[16] Kushal Babel, Nerla Jean-Louis, Yan Ji, Ujval Misra, Mahimna Kelkar, Kosala Yapa Mudiyanselage, Andrew Miller, and Ari Juels. PROF: protected order flow in a profit-seeking world. *CoRR*, abs/2408.02303, 2024.

[17] Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. Smart contract languages: A comparative analysis. *Future Gener. Comput. Syst.*, 164:107563, 2025.

[18] Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. Smart contract languages: A comparative analysis. *Future Generation Computer Systems*, 164:107563, 2025.

[19] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. Maximizing extractable value from Automated Market Makers. In *Financial Cryptography*, volume 13411 of *LNCS*, pages 3–19. Springer, 2022.

[20] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of Automated Market Makers in DeFi. *Logical Methods in Computer Science*, 18(4), 2022.

[21] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Fun with Bitcoin smart contracts. In *ISoLA*, pages 432–449, 2018.

[22] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Log. Methods Comput. Sci.*, 17(4), 2021.

[23] Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive Bitcoin contracts. *Log. Methods Comput. Sci.*, 18(1), 2022.

[24] Massimo Bartoletti, Stefano Lande, and Roberto Zunino. Bitcoin covenants unchained. In *ISoLA*, pages 432–449, 2020.

[25] Massimo Bartoletti, Stefano Lande, and Roberto Zunino. Computationally sound bitcoin tokens. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[26] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. DeFi composability as MEV non-interference. In *Financial Cryptography*, volume 14744 of *LNCS*. Springer, 2024.

[27] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor utxo blockchains. In *2024 IEEE 9th European Symposium on Security and Privacy*, pages 235–267, 2024.

[28] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Certifying optimal mev strategies with lean, 2025.

[29] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Disco_sim: a simulator for a hybrid utxo-/account- based blockchain supporting truly distributed smart contracts, 2025.

[30] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Scalable utxo smart contracts via fine-grained distributed state. *Future Generation Computer Systems*, 175:108023, 2026.

[31] Massimo Bartoletti and Roberto Zunino. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[32] Massimo Bartoletti and Roberto Zunino. BitML: a calculus for Bitcoin smart contracts. In *ACM CCS*, 2018.

[33] Massimo Bartoletti and Roberto Zunino. A theoretical basis for MEV. In *Financial Cryptography and Data Security*, LNCS. Springer, 2025.

[34] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. SoK: Mitigation of Front-Running in Decentralized Finance. In *Financial Cryptography Workshops*, volume 13412 of *LNCS*, pages 250–271. Springer, 2022.

[35] Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: privacy-preserving decentralized cryptocurrency exchange. In

Applied Cryptography and Network Security (ACNS), volume 12726 of *LNCS*, pages 163–194. Springer, 2021.

[36] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[37] Eike Best and Raymond R. Devillers. Sequential and concurrent behaviour in petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.

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

[39] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2020.

[40] Lorenz Breidenbach, Philip Daian, Florian Tramèr, and Ari Juels. Enter the Hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *USENIX Security Symposium*, pages 1335–1352. USENIX Association, 2019.

[41] Lars Brünjes and Murdoch James Gabbay. UTxO- vs account-based smart contract blockchain programming paradigms. In *ISoLA*, volume 12478 of *LNCS*, pages 73–88. Springer, 2020.

[42] Andrea Canidio and Vincent Danos. Commitment against front-running attacks. *Manag. Sci.*, 70(7):4429–4440, 2024.

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

[44] Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio Mijail Anton Quiles. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *Formal Methods (FM)*, volume 14000 of *LNCS*, pages 571–583. Springer, 2023.

[45] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. UTXO_{ma}: UTXO with multi-asset support. In *ISoLA*, volume 12478 of *LNCS*, pages 112–130. Springer, 2020.

[46] Manuel M.T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTXO model. In *Workshop on Trusted Smart Contracts*, 2020.

[47] Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. Analyzing and benchmarking ZK-rollups. Cryptology ePrint Archive, Paper 2024/889, 2024.

[48] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. FairMM: A fast and frontrunning-resistant crypto market-maker. In *Cyber Security, Cryptology, and Machine Learning (CSCML)*, volume 13301 of *LNCS*, pages 428–446. Springer, 2022.

[49] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is Solidity Solid Enough? In *Workshop on Trusted Smart Contracts (WTSC)*, volume 11599 of *LNCS*, pages 138–153. Springer, 2019.

[50] 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*, pages 910–927. IEEE, 2020.

[51] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FASTKITTEN: practical smart contracts on bitcoin. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 801–818, USA, 2019. USENIX Association.

[52] Andrea De Salve, Luca Franceschi, Andrea Lisi, Paolo Mori, and Laura Ricci. L2DART: A trust management system integrating blockchain

and off-chain computation. *ACM Trans. Internet Technol.*, 23(1), feb 2023.

[53] Christian Decker and Rusty Russell. eltoo : A simple layer 2 protocol for Bitcoin, 2018.

[54] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 9212 of *LNCS*, pages 3–18. Springer, 2015.

[55] Marco Dessalvi. Automated Market Makers in the Lean 4 Theorem Prover. Master's thesis, DTU Compute, Technical University of Denmark, 2025.

[56] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 303–312. ACM, 2017.

[57] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Bulletin of the EATCS*, 124, 2018.

[58] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283, 1968.

[59] Cayo Fletcher-Smith and Muntadher Sallal. Security analysis of blockchain layer-one sharding based Extended-UTxO model. In *Communications, Networking, and Information Systems*, pages 95–123. Springer, 2023.

[60] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2315–2326. ACM, 2022.

[61] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 232–244. ACM, 2023.

[62] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust (POST)*, volume 10804 of *LNCS*, pages 243–269. Springer, 2018.

[63] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 201–226, Cham, 2020. Springer International Publishing.

[64] Kevin Hammond. Plutus fee estimator: find out the cost of transacting on Cardano. https://iohk.io/en/blog/posts/2022/01/21/plutus-fee-estimator-find-out-the-cost-of-transacting-on-cardano/, 2022.

[65] Lioba Heimbach and Roger Wattenhofer. Sok: Preventing transaction reordering manipulations in decentralized finance. In *Advances in Financial Technologies*, 2022.

[66] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE Computer Society, 2018.

[67] Paul E. Hoffman. DNS Security Extensions (DNSSEC). RFC 9364, February 2023.

[68] IOHK. How to write a scalable Plutus app – UTXO congestion. https://plutus-apps.readthedocs.io/en/latest/plutus/howtos/writing-a-scalable-app.html#utxo-congestion, 2022.

[69] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 2020.

[70] Michael Peyton Jones. Inline datums, 2021. https://cips.cardano.org/cip/CIP-32.

[71] Michael Peyton Jones. Reference inputs, 2021. https://cips.cardano.org/cip/CIP-31.

[72] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*, 2018.

[73] Md Monjurul Karim, Dong Hoang Van, and Qiang Qu. Securing decentralized finance: A comprehensive survey of maximal extractable value and its countermeasures. *Blockchain: Research and Applications*, page 100455, 2026.

[74] Andre Knispel, James Chapman, Joosep Jääger, Ulf Norell, Orestis Melkonian, Alasdair Hill, and William DeMeo. Formal specification of the Cardano blockchain ledger, mechanized in Agda. In *Workshop on Formal Methods for Blockchains (FMBC)*, 2024.

[75] Kshitij Kulkarni, Theo Diamandis, and Tarun Chitra. Routing MEV in Constant Function Market Makers. In *Web and Internet Economics (WINE)*, volume 14413 of *LNCS*, pages 456–473. Springer, 2023.

[76] Chao Li, Balaji Palanisamy, and Runhua Xu. Scalable and privacy-preserving design of on/off-chain smart contracts. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 7–12, 2019.

[77] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Xi Chen, and Ting Chen. Demystifying DeFi MEV Activities in Flashbots Bundle. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 165–179. ACM, 2023.

[78] Eric Lombrozo, Johnson Lau, and Pieter Wuille. Segregated witness (consensus layer), 2015. BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[79] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[80] Dario Maddaloni, Riccardo Marchesin, and Roberto Zunino. How to save fees in bitcoin smart contracts: a simple optimistic off-chain protocol. *Blockchain: Research and Applications*, page 100365, 2025.

[81] Diego Marmsoler and Achim D. Brucker. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. *Formal Aspects Comput.*, 37(2):15:1–15:56, 2025.

[82] Bruno Mazorra, Michael Reynolds, and Vanesa Daza. Price of MEV: towards a game theoretical approach to MEV. In *ACM CCS Workshop on Decentralized Finance and Security*, pages 15–22. ACM, 2022.

[83] Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 126–141. Springer, 2016.

[84] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction (CADE)*, page 625–635. Springer-Verlag, 2021.

[85] Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Reality-based UTXO ledger. *Distributed Ledger Technol. Res. Pract.*, 2(3):1–33, 2023.

[86] Jonas Nick, Andrew Poelstra, and Gregory Sanders. Liquid: a Bitcoin sidechain. https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf, 2020.

[87] Russell O'Connor. Simplicity: A new language for blockchains. In *PLAS*, 2017.

[88] Russell O'Connor and Marta Piekarska. Enhancing Bitcoin transactions with covenants. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[89] Andreas Park. The conceptual flaws of decentralized automated market making. *Management Science*, 69(11):6731–6751, 2023.

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

[91] Tim Ruffing Pieter Wuille, Jonas Nick. Schnorr Signatures for secp256k1, 2020. BIP 340, https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

[92] George Pîrlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1327–1341. ACM, 2021.

[93] Plutus Team. Formal specification of the PlutusCore language, 2022.

[94] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments, 2015.

[95] Daniele Pusceddu and Massimo Bartoletti. Formalizing Automated Market Makers in the Lean 4 Theorem Prover. In *Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[96] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *IEEE Symp. on Security and Privacy*, pages 198–214. IEEE, 2022.

[97] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the DeFi ecosystem with Flash Loans for fun and profit. In *Financial Cryptography*, volume 12674 of *LNCS*, pages 3–32. Springer, 2021.

[98] Geoffrey Ramseyer and David Mazières. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *CoRR*, abs/2404.03201, 2024.

[99] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 12–27. ACM Press, 1988.

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

[101] Alejo Salles. On the formalization of MEV, 2021. https://writings.flashbots.net/research/formalization-mev.

[102] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in Ethereum smart contracts. In *International Conference on Blockchain Economics, Security and Protocols (Tokenomics)*, volume 71 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[103] sBTC working group. Stacks: a Bitcoin Layer for Smart Contracts, 2023.

[104] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019.

[105] Solana Foundation. Solana documentation: Transactions and instructions, 2024. https://solana.com/docs/core/transactions#array-of-account-addresses.

[106] Solidity Academy. #100DaysOfSolidity #073: Understanding denial of service attacks in Solidity smart contracts, 2023.

[107] Alvise Spanò, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Dalila Ressi, and Sabina Rossi. Algomove: Typed abstractions for algorand smart contracts. *Blockchain: Research and Applications*, page 100485, 2026.

[108] SUI Foundation. All about parallelization. https://blog.sui.io/parallelization-explained, 2024.

[109] Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy Journal of Transhuman Thought, 16*, pages 50–53, 1996.

[110] The mathlib Community. The Lean Mathematical Library. In *Certified Programs and Proofs (CPP)*, page 367–381. ACM, 2020.

[111] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022.

[112] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner Jones and the Raiders of the Dark Forest: An empirical study of frontrunning on the Ethereum blockchain. In *USENIX Security Symposium*, pages 1343–1359, 2021.

[113] Domenico Tortola, Andrea Lisi, Paolo Mori, and Laura Ricci. Tethering layer 2 solutions to the blockchain: A survey on proving schemes. *Computer Communications*, 225:289–310, 2024.

[114] Jianhuan Wang, Jichen Li, Zecheng Li, Xiaotie Deng, and Bin Xiao. n-MVTL Attack: Optimal Transaction Reordering Attack on DeFi. In *European Symposium on Research in Computer Security (ESORICS)*, volume 14346 of *LNCS*, pages 367–386. Springer, 2023.

[115] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. SoK: Decentralized Finance (DeFi). In *ACM Conference on Advances in Financial Technologies, (AFT)*, pages 30–46. ACM, 2022.

[116] Pieter Wuille and Andrew Poelstra. Miniscript: Streamlined Bitcoin scripting. https://medium.com/blockstream/miniscript-bitcoin-scripting-3aeff3853620, 2019.

[117] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proc. VLDB Endow.*, 14(11):2314–2326, 2021.

[118] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Comput. Surv.*, 55(11), 2023.

[119] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. 2017.

[120] Haowen Zhang, Jing Li, He Zhao, Tong Zhou, Nianzu Sheng, and Hengyu Pan. BlockPilot: A proposer-validator parallel execution framework for blockchain. In *International Conference on Parallel Processing (ICPP)*, pages 193–202. ACM, 2023.

[121] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-Frequency Trading on Decentralized On-Chain Exchanges. In *IEEE Symp. on Security and Privacy*, pages 428–445. IEEE, 2021.