Università degli Studi di Cagliari

Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Matematica e Informatica
Ciclo XXXIII

Ph.D. Thesis

Formal Methods for Secure Bitcoin Smart Contracts

S.S.D. INF/01

Candidate
Stefano Lande

Supervisor
Prof. Massimo Bartoletti

PhD Coordinator
Prof. Michele Marchesi
Prof. Roberto Tonelli

Academic year 2019 - 2020
Thesis defended in April 2021

Acknowledgments

Stefano Lande gratefully acknowledges Sardinian Regional Government for the financial support of his PhD scholarship (P.O.R. Sardegna F.S.E. - Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2014-2020 - Axis III Education and training, Thematic goal 10, Investment Priority 10ii), Specific goal 10.5.

UNIONE EUROPEA
Fondo sociale europeo

REGIONE AUTONOMA DELLA SARDEGNA

FSE 2007-2013
POR
SARDEGNA
obiettivo competitività regionale e occupazione

Abstract

The notion of *smart contracts* was introduced in 1997 by Nick Szabo, to describe agreements among mutually distrusting parties that can be automatically enforced without resorting to a trusted intermediary. Then, the idea was mostly forgotten due to the technical impossibility to implement it. The advent of distributed ledger technologies, pioneered by Bitcoin, provided a technical foundation to reshape and develop smart contracts.

Since smart contracts handle the ownership of valuable assets, attackers may be tempted to exploit vulnerabilities in their implementation to steal or tamper with these assets. For instance, a series of vulnerabilities in Ethereum contracts have been exploited, causing money losses in the order of hundreds of millions of dollars.

Over the last years, a variety of smart contracts for Bitcoin have been proposed, both by the academic community and by that of developers. However, the heterogeneity in their treatment, the informal (often incomplete or imprecise) descriptions, and the use of poorly documented Bitcoin features, poses obstacles to the development of secure smart contracts. Using formal models and domain-specific languages to describe the behaviour of the underlying platform, and to model contracts, could help to overcome these security issues, by reducing the distance between the intended behaviour of a contract and the implementation.

In this thesis, we propose a formal model of Bitcoin transactions, which is the foundation for a new process algebra for defining Bitcoin smart contracts. Furthermore, we present a toolchain for developing smart contracts in BitML, a domain-specific language based on the contributions of this thesis. Moreover, we propose a new extension to Bitcoin, called neighbourhood covenants, which extends its expressiveness as a smart contract platform. We then exploit neighbourhood covenants to implement fungible tokens on Bitcoin.

Contents

Introduction

In recent years we have observed a growing interest around *cryptocurrencies*. Bitcoin [116], the first decentralized cryptocurrency, was introduced in 2009, and through the years it has consolidated its position as the most popular one. Bitcoin and other cryptocurrencies have pushed forward the concept of decentralization, providing means for reliable interactions between mutually distrusting parties on an open network.

Besides the intended monetary application, the Bitcoin blockchain can be seen as a way to consistently maintain the state of a system over a peer-to-peer network, without the need of a trusted authority. If the system is a currency, its state is the amount of funds in each account. This concept can be generalised to the case where the system is a *smart contract* [72], namely an executable computer protocol which can also handle transfers of currency. The idea of exploiting the Bitcoin blockchain to build smart contracts has recently been explored by several works. Lotteries [4, 26, 62, 23], gambling games [57], contingent payments [12], covenants [63, 66], and other kinds of fair computations [1, 56] are some examples of the capabilities of Bitcoin as a platform for smart contracts.

The term "smart contract" was conceived by Nick Szabo [72] to describe agreements between two or more parties, that can be automatically enforced without a trusted intermediary. Fallen into oblivion for several years, the idea of smart contract has been resurrected with the recent surge of distributed ledger technologies, led by Ethereum [98] and Hyperledger [109]. In such incarnations, smart contracts are rendered as computer programs. Users can request the execution of contracts by sending suitable *transactions* to the nodes of a peer-to-peer network. These nodes collectively maintain the history of all transactions in a public, append-only data structure, called *blockchain*. The sequence of transactions on the blockchain determines the state of each contract, and, accordingly, the assets of each user.

A crucial feature of smart contracts is that their correct execution does *not* rely on a trusted authority: rather, the nodes which process transactions are assumed to be mutually untrusted. Potential conflicts in the execution of contracts are resolved through a *consensus* protocol, whose nature depends on the specific platform (e.g., it is based on "proof-of-work" in Ethereum). Ideally, contracts execute correctly whenever the adversary does not control the majority of some resource (e.g., computational power for "proof-of-work" consensus).

Since smart contracts handle the ownership of valuable assets, attackers may be tempted to exploit vulnerabilities in their implementation to steal or tamper with these assets. Although analysis tools [60, 27, 52] may improve the security of contracts, so far they have not been able to completely prevent attacks. For instance, a series of vulnerabilities in Ethereum contracts [8]) have been exploited, causing money losses in the order of hundreds of millions of dollars [124, 120, 77].

Unlike Ethereum, where contracts can be expressed as computer programs with a well-defined semantics [51, 125], Bitcoin contracts are usually realised as cryptographic protocols relying on features of Bitcoin that go beyond the standard transfers of currency. Roughly, participants of the protocols send/receive messages, verify signatures, and put/search custom-designed transactions on the blockchain. While the vast majority of Bitcoin transactions uses scripts only to verify signatures, the transactions of smart contracts exploit more complex scripts, e.g. to determine the winner of a lottery, or to check if a secret has been revealed. Smart contracts may also exploit other (infrequently used) features of Bitcoin, e.g. various signature modifiers, and temporal constraints on transactions.

As a matter of fact, using these advanced features to design a new smart contract is not a trivial matter, for two reasons. First, while the overall behaviour of Bitcoin is clear, the details of many of its crucial aspects are poorly documented. To understand the details of how a mechanism actually works, one has to explore various web pages (often inaccurate, or inconsistent, or overly technical), and eventually resort to the source code of the Bitcoin client [83] to have the correct answer. Second, the description of advanced features is often too concrete to be effectively used in the design and analysis of a smart contract (indeed, in many cases the only available description coincides with the implementation).

The informal (often incomplete or imprecise) narration of these protocols, together with the use of poorly documented features of Bitcoin (e.g., segregated witnesses, scripts, signature modifiers, temporal constraints), and the overall heterogeneity in their treatment, pose serious obstacles to the research on smart contracts in Bitcoin.

Using formal models and and domain-specific languages (possibly, not

Turing-complete) to describe the behaviour of the underlying platform and to model contracts could help to overcome these security issues. by reducing the distance between contract specification and implementation.

Contributions

Bitcoin formalization We propose a formal model of Bitcoin transactions. This model is abstract enough to allow for formal reasoning on the behaviour of Bitcoin transactions. For instance, we use our model to formally prove some properties of the Bitcoin blockchain, e.g. that transactions cannot be spent twice (Theorem 3.1.2), and that the overall value contained in the blockchains (excluding the coinbase transactions) is decreasing (Theorem 3.1.5).

Our model formally specifies some poorly documented features of Bitcoin, e.g. transaction signatures and signature modifiers (Definition 3.5), output scripts (Definitions 3.1 and 3.9), multi-signature verification (Definition 3.8), Segregated Witnesses (Definitions 3.3 and 3.11), paving the way towards automatic verification.

Then, we provide the first systematic survey of smart contracts on Bitcoin. In order to obtain a uniform and precise treatment, we propose a new formal model of contracts, which exploits the previous one about Bitcoin transactions. This model is based on a process calculus with primitives to construct Bitcoin transactions, to put them on the blockchain, and to search the blockchain for transactions matching given patterns. Our calculus allows us to give smart contracts a precise operational semantics, which describes the interactions of the (possibly dishonest) participants involved in the execution of a contract.

We systematically formalise a large portion of the contracts proposed so far both by researchers and Bitcoin developers. In many cases, we find that specifying a contract with the intended security properties is significantly more complex than expected after reading the informal descriptions of the contract. Usually, such informal descriptions focus on the case where all participants are honest, neglecting the cases where one needs to compensate for some unexpected behaviour of the dishonest environment.

New Bitcoin extensions We propose a variant of covenants, named *neighbourhood covenants*, which can inspect not only the redeeming transaction, but also the siblings and the parent of the spent one. This extension preserves the basic UTXO design of Bitcoin, adding only a few opcodes to its script language, which is kept efficient, loop-free, and *non*

Turing-complete. Still, neighbourhood covenants significantly increase the expressiveness of Bitcoin as a smart contracts platform, allowing to execute arbitrary smart contracts by appending a *chain* of transactions to the blockchain. Technically, we prove that neighbourhood covenants make Bitcoin Turing-complete.

We summarise the contribution as follows:

- we propose neighbourhood covenants as a Bitcoin extension (Section 4.1), and we show that they make Bitcoin Turing-powerful (Theorem 4.1.1). We then discuss how to efficiently implement them on Bitcoin (Section 4.4);
- we use our formal model to specify complex Bitcoin contracts, which largely extend the set of use cases expressible in pure Bitcoin.
- we show how this form of covenants can be exploited in an high-level language like BitML.

A toolchain for Developing BitML contract We consider BitML, a high-level language for smart contracts, featuring a computationally sound embedding into Bitcoin [22], and a sound and complete verification technique of relevant trace properties [24]. BitML can express many of the smart contracts appeared in the literature [15, 9], and execute them by appending suitable transactions to the Bitcoin blockchain.

We develop a toolchain for writing and verifying BitML contracts, and for deploying them on Bitcoin. More specifically, our contribution can be summarised as follows:

- A BitML embedding in Racket [42], which allows for programming BitML contracts within the DrRacket IDE.
- A security analyzer which can check arbitrary LTL properties of BitML contracts. In particular, the analysis can decide *liquidity*, a landmark property of smart contracts. requiring that the funds within a contract do not remain frozen forever.
- A compiler from BitML contracts to standard Bitcoin transactions.
- A collection of BitML contracts, which we use as a benchmark to evaluate our toolchain. This collection contains some of the most complex contracts ever developed for Bitcoin, e.g. financial services, auctions, timed commitments, lotteries, and a variety of other gambling games. We use our benchmarks to discuss the expressiveness and the limitations of Bitcoin as a smart contracts platform.

The toolchain is open-source and its component are available at [95].

On-chain Fungible Tokens on Bitcoin Bitcoin, due to the limitation of its simple scripting language, does not support fungible tokens directly. Nevertheless, the first implementations of tokens were developed before Ethereum, on top of Bitcoin. All these implementations have a common drawback: the correctness of the token actions is *not* guaranteed by the consensus protocol of the blockchain. In fact, the blockchain is used just to notarize the actions that manipulate tokens, but not to check that these actions are actually permitted.

By contrast, modern blockchain platforms support *on-chain* tokens, whose correctness is guaranteed by the consensus protocol of the blockchain. Since adding native tokens to Bitcoin appears to be out of reach, we exploit neighborhood covenants, which increases the expressiveness of Bitcoin enough to support tokens. Summarising:

- we introduce a symbolic model of fungible tokens, which formalises their archetypal features: their minting and burning, the split and join operations, and the exchange of tokens with other tokens or with bitcoins (Section 7.2);
- we exploit neighbourhood covenants to implement tokens on Bitcoin (Section 7.3);

Structure of the thesis

We briefly describe the overall structure this thesis below.

Chapter 1: Blockchain and Smart Contracts overviews blockchain technologies and Bitcoin in particular. Part of this material borrows from [11, 7].

Chapter 2: BitML: a calculus for Bitcoin smart contracts briefly overviews BitML, an high-level language for specifying smart contracts which can be executed on Bitcoin. Then, it gives an intuition of the notion of liquidity of a contract, and its verification technique. Part of this material borrows from [24].

Chapter 3: A Formal Model of Bitcoin Transactions proposes a formal model of Bitcoin transactions, which is sufficiently abstract to enable formal reasoning, and at the same time is concrete enough to serve as an alternative documentation to Bitcoin. This chapter is based on [11].

Chapter 4: Extending Bitcoin with Neighborhood Covenants proposes a formal model of covenants, i.e. linguistic primitives that allow transactions to constrain the scripts of the redeeming ones. Our proposal increases the expressivity of traditional covenants by allowing the spending conditions in a transaction to depend on the neighbour transactions We use our model to specify some complex Bitcoin contracts, and we discuss how to exploit covenants to design high-level language primitives for Bitcoin contracts. This chapter is based on [18].

Chapter 5: Bitcoin Smart Contracts as Endpoint Protocols presents a comprehensive survey of smart contracts on Bitcoin, in a uniform framework. Our treatment is based on a new formal specification language for smart contracts, which also helps us to highlight some subtleties in existing informal descriptions, making a step towards automatic verification. We discuss some obstacles to the diffusion of smart contracts on Bitcoin, and we identify the most promising open research challenges. This chapter is based on [9].

Chapter 6: A Toolchain for Developing BitML Contracts present a BitML toolchain for developing and verifying smart contracts that can be executed on Bitcoin. Our toolchain translates BitML contracts into sets of standard Bitcoin transaction, and automatically verifies relevant properties of contracts. This chapter is based on [10].

Chapter 7: On-chain Fungible Tokens on Bitcoin proposes a secure and efficient implementation of fungible tokens on Bitcoin, based on neighbourhood covenants. We propose a symbolic model for fungible token which can be applied to UTXO-based blockchains, and prove some relevant security properties. This chapter is based on [19].

Conclusions summarises our work and shows a comparison with the related ones. Finally, it outlines some future perspectives and extensions.

Part I

Background

Chapter 1

Blockchain and Smart Contracts

A blockchain is an immutable data structure that maintains an ordered set of *transactions*. For implementation reasons, transactions are grouped into blocks. Each block is chained together with the previous one by including its hash. Changing the past, i.e. altering the data contained in a block, would change its hash, consequently invalidating the chain. The blockchain is distributed and maintained by a network of nodes: each node maintain a local copy of the blockchain and use a *consensus mechanism* that ensures that each node agrees on the new block to append.

The process of append a new block is called *mining*. A subset of nodes, called *miners*, try to build a new block to append to the blockchain. To avoid conflicts and regulate this process, miners reach consensus on the blockchain by voting. In Bitcoin and Ethereum, the vote is done though *Proof of Work*, i.e. a miner must solve a cryptographic puzzle that requires time-consuming computations. Conceptually, the more computing power a miner has, the more voting power she has on deciding the next block In other blockchain platforms, like Algorand, miners voting power is proportional to the amount of currency they own. This mechanism is called *Proof of Stake*.

The purpose of a transaction depends on the blockchain platform, although the main goal is usually to exchange a currency. In Bitcoin, transactions register a transfer of bitcoins between users, albeit alternative uses have been studied to store arbitrary data on the blockchain [21, 17].

This chapter provides an overview of Bitcoin. Section 1.1 explains its main aspects, detailed enough to understand the formal model of Bitcoin

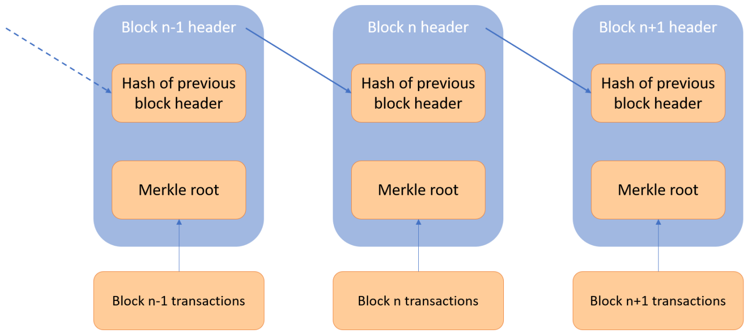


Figure 1.1: *Simplified Blockchain.*

transactions and its extension proposed in Chapters 3 and 4, and the model of Bitcoin contracts as protocols in Chapter 5.

1.1 Bitcoin

Bitcoin [116], is the first decentralized cryptocurrency. The nodes of the Bitcoin network maintain a public and immutable data structure, called *blockchain*. The blockchain stores the historical record of all transfers of bitcoins, which are referred to as *transactions*. When a node updates the blockchain, the other nodes verify if the appended transactions are valid, e.g. by checking if the conditions specified in *scripts* are satisfied. Scripts are programmable boolean functions: in their standard (and mostly used) form they verify a digital signature against a public key. Since the blockchain is immutable, tampering with a stored transaction would result in the invalidation of all the subsequent ones. Updating the state of the blockchain, i.e. appending new transactions, requires solving a moderately difficult cryptographic puzzle. In case of conflicting updates, the chain that required the largest computational effort is considered the valid one. Hence, the immutability and the consistency of the blockchain is bounded by the total computational power of honest nodes. An adversary with enough resources can append invalid transactions, e.g. with incorrect digital signatures, or rewrite a part of the blockchain, e.g. to perform a *double-spending attack*. The attack consists in paying someone by publishing a transaction on the blockchain, and then removing it (making the funds unspent).

1.1.1 Transactions

Users interact with Bitcoin through *addresses*, which they can freely generate. *Transactions* describe transfers of bitcoins (Ƀ) between addresses. The log of all transactions is recorded on a public, immutable and decentralised data structure called *blockchain*. To explain how the blockchain works, consider the transactions T_0 and T_1 displayed in Figure 1.2.

T_0
in: $\cdots$
wit: $\cdots$
out: $(\lambda x.\mathsf{versig}(k, x), v_0$Ƀ$)$

T_1
in: T_0
wit: σ
out: $(\lambda y.e, v_1$Ƀ$)$

Figure 1.2: *Two Bitcoin transactions.*

The transaction T_0 contains v_0Ƀ, which can be *redeemed* by putting on the blockchain a transaction (e.g., T_1), whose in field is a reference to T_0. To redeem T_0, the *witness* of the redeeming transaction (the value in its wit field) must make the *output script* of T_0 (the first element of the pair in the out field) evaluate to true. When this happens, the value of T_0 is transferred to the new transaction, and T_0 is no longer redeemable.

In the example displayed before, the output script of T_0 evaluates to true when receiving a digital signature on the redeeming transaction T_1, with a given key pair k. We denote with $\mathsf{versig}(k, x)$ the verification of the signature x on the redeeming transaction: of course, since the signature must be included in the witness of the redeeming transaction, it will consider all the parts of that transaction *except* its wit field. We assume that σ is the signature of T_1, computed with the key pair k.

Now, assume that the blockchain contains T_0, not yet redeemed, and someone tries to append T_1. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the output script of T_0, by instantiating its formal parameter x to the signature σ in the witness of T_1. The function $\mathsf{versig}(k, \sigma)$ verifies that σ is actually the signature of T_1: therefore, the output script succeeds, and T_1 redeems T_0. Subsequently, a new transaction can redeem T_1 by satisfying its output script $\lambda y.e$ (not specified in the figure). The formalism used to representT_0 and T_1 is fully presented in Chapter 3.

Bitcoin transactions may be more general than the ones illustrated by the previous example. First, there can be multiple inputs and outputs. Each output has an associated output script and value, and can be redeemed independently from the others. Consequently, in fields must specify which output they are redeeming. A transaction with multiple inputs associates a witness to each of them. The sum of the values of all the inputs must be greater or equal to the sum of the values of all the outputs, otherwise the transaction is considered invalid. In its general form, the output script is a program in a (non Turing-complete) scripting language, featuring a limited set of logic, arithmetic, and cryptographic operators. Finally, a transaction can specify time constraints (absolute, or relative to its input transactions) about when it can appear on the blockchain.

1.1.2 Smart contracts

Albeit the primary usage of Bitcoin is to exchange currency, its blockchain and consensus mechanism can also be exploited to securely execute some forms of *smart contracts*. These are agreements among mutually distrusting parties, which can be automatically enforced without resorting to a trusted intermediary.

Bitcoin scripting language permits the definition of several smart contracts. Differently from Ethereum, where smart contracts are long-lived programs stored and invoked on the blockchain through transactions, in Bitcoin they are modelled as cryptographic protocols that may spread multiple transactions, they have no state, and cannot be reused once terminated.

A new process algebra to express smart contracts in Bitcoin is presented in Chapter 5. This formalism is expressive enough to model real use-cases in the Bitcoin community, providing a clear semantics and enabling formal reasoning. Briefly, the participants involved in a smart contract create and publish new transactions on the blockchain and interact with other participants to exchange signatures.

Chapter 2

BitML: a calculus for Bitcoin smart contracts

In this section we briefly overview BitML, a domain-specific language for Bitcoin smart contracts. BitML is a process calculus, with primitives to stipulate contracts and to exchange currency according to the contract terms. In this respect, BitML departs from the current practice of representing Bitcoin contracts as cryptographic protocols: rather, BitML pioneers the "contracts-as-program" paradigm for Bitcoin, by completely abstracting from Bitcoin transactions and cryptographic details. We then give some intuition about liquidity and our verification technique.

We assume a set of *participants*, ranged over by $\mathsf{A}, \mathsf{B}, \ldots$, and a set of names, of two kinds: $x, y, \ldots$ denote *deposits* of ₿, while $a, b, \ldots$ denote *secrets*. We write $\boldsymbol{x}$ (resp. $\boldsymbol{a}$) for a finite sequence of deposit (resp. secrets) names.

2.1 BitML in a nutshell

BitML allows participants to exchange cryptocurrency according to pre-agreed contract rules. In BitML, any participant can broadcast a *contract advertisement* $\{G\}C$, where C is the actual contract, specifying the rules to transfer bitcoins (₿), while G is a set of *preconditions* to its execution.

Preconditions (Figure 2.1) may require participants to deposit some ₿ in the contract (either upfront or at runtime), or to commit to some secret. More in detail, $\mathsf{A} : \texttt{!}\, v \,\texttt{@}\, x$ requires A to own v₿ in a deposit x, and to spend it for stipulating a contract C. Instead, $\mathsf{A} : \texttt{?}\, v \,\texttt{@}\, x$ only requires A to pre-authorize the spending of x, which can be gathered by the contract

$G ::= \mathsf{A}\texttt{:?}\,v\,\texttt{@}\,x$	volatile deposit of vɃ, expected from A
$\mid \mathsf{A}\texttt{:!}\,v\,\texttt{@}\,x$	deposit of vɃ put by A
$\mid \mathsf{A}\texttt{:secret}\,a$	secret committed by A
$\mid G \mid G$	composition

Figure 2.1: *Syntax of BitML contract preconditions.*

$C ::= \sum_{i\in I} D_i$	contract
$D ::=$	guarded contract
$\texttt{put}\,\boldsymbol{x}\,\&\,\texttt{reveal}\,\boldsymbol{a}\,\texttt{if}\,p.\,C$	put deposits and reveal secrets (if p is true)
$\mid \texttt{withdraw}\,\mathsf{A}$	transfer the balance to A
$\mid \texttt{split}\,\boldsymbol{v} \to \boldsymbol{C}$	split the balance
$\mid \mathsf{A} : D$	wait for A's authorization
$\mid \texttt{after}\,t : D$	wait until time t

$p ::=$	predicate		
true	truth	$E ::=$	contract expression
$\mid p \wedge p$	conjunction	a	secret
$\mid \neg p$	negation	$\mid E + E$	addition
$\mid E = E$	equality	$\mid E - E$	subtraction
$\mid E < E$	less than		

Figure 2.2: *Syntax of BitML contracts.*

at run-time. The precondition $\mathsf{A}\texttt{:secret}\,a$ requires A to commit to a secret a before C starts.

After $\{G\}C$ has been advertised, each participant can choose whether to accept it, or not. When all the preconditions G have been satisfied, and all the involved participants have accepted, the contract C becomes *stipulated*. The contract starts its execution with a balance, initially set to the sum of the `!`-deposits required by its preconditions. Running C will affect this balance, when participants deposit/withdraw funds to/from the contract.

A contract C (Figure 2.2) is a *choice* among zero or more branches. Each branch is a *guarded contract* which enables an action, and possibly proceeds with a continuation C'. The guarded contract $\texttt{withdraw}\,\mathsf{A}$ transfers the whole balance to A, while $\texttt{split}\; v_1 \to C_1 \mid \cdots \mid v_n \to C_n$ decomposes the contract into n parallel components C_i, each one with balance v_i. The guarded contract $\texttt{put}\, \boldsymbol{x} \,\&\, \texttt{reveal}\, \boldsymbol{a}\, \texttt{if}\, p$ atomically performs the following: (i) spend all the ?-deposits $\boldsymbol{x}$, adding their values to the contract balance; (ii) check that all the secrets $\boldsymbol{a}$ have been revealed and satisfy the predicate p. When enabled, the above-mentioned actions can be fired by anyone, at anytime. To restrict *who* can execute actions and *when*, one can use the decoration $\mathsf{A} : D$, which requires the authorization of A, and the decoration $\texttt{after}\, t : D$, which requires to wait until time t.

2.1.1 A basic example

As a first example, we express in BitML the *timed commitment* [4], a basic protocol to construct more complex contracts, like e.g. lotteries and other games [5]. In the timed commitment, a participant A wants to choose a secret, and promises to reveal it before some time t. The contract ensures that if A does not reveal the secret in time, then she will pay a penalty of 1₿ to B (e.g., the opponent player in a game). In BitML, this is modelled as follows:

$$\{\mathsf{A}\texttt{:!}\,1\,\texttt{@}\,x \mid \mathsf{A}\texttt{:secret}\, a\}\ (\texttt{reveal}\, a.\texttt{withdraw}\ \mathsf{A}\ \ +\ \ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B})$$

The precondition requires A to pay upfront 1₿, and to commit to a secret a. The contract is a non-deterministic choice between two branches. Only A can choose the first branch, by performing $\texttt{reveal}\, a$ (syntactic sugar for $\texttt{put}\, \varepsilon \,\&\, \texttt{reveal}\, a\, \texttt{if}\, \mathit{true}$, where ε is the empty sequence). Subsequently, anyone can transfer 1₿ to A. Only after t, if the `reveal` has not been fired, any participant can fire `withdraw` B in the second branch, moving 1₿ to B. So, before t, A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). If no branch is taken by t, the first one who fires its `withdraw` gets 1₿.

2.2 BitML semantics

We briefly recall from [22] the semantics of BitML. The semantics is a labelled transition system between configurations of the following form:

- $\{G\}C$, representing the advertisement of contract C with preconditions G;
- $\langle C, v\rangle_x$, representing a stipulated contract, holding a current balance of v₿. The name x uniquely identifies the contract in a configuration;

- $\langle \mathsf{A}, v\rangle_x$ representing a fund of vB owned by A, and with unique name x;
- $\mathsf{A} \rhd \chi$, representing A's *authorizations* to perform some operation χ. We refer to [22] for the syntax of authorizations (some of them are exemplified below);
- $\{\mathsf{A} : a\#N\}$, representing that A has committed to a random secret a with (secret) length N;
- $\mathsf{A} : a\#N$, representing that A has revealed her secret a (with its length N).
- $\Gamma \mid \Delta$ is the parallel composition of two configurations (with identity $\mathbf{0}$);
- $\Gamma \mid t$ is a *timed* configuration, where $t \in \mathbb{N}$ is a global time.

We now illustrate the BitML semantics by examples; when time is immaterial, we only show the steps of the untimed semantics. We omit labels on transitions.

Deposits. When A owns a deposit $\langle \mathsf{A}, v\rangle_x$, she can use it in various ways: she can divide the deposit into two smaller deposits, or join it with another deposit of hers to form a larger one; the deposit can also be transferred to another participant, or destroyed. For instance, to donate a deposit x to B, A must first issue the authorization $\mathsf{A} \rhd x \rhd \mathsf{B}$; then, anyone can transfer the money to B:

$$\langle \mathsf{A}, v\rangle_x \mid \cdots \;\rightarrow\; \langle \mathsf{A}, v\rangle_x \mid \mathsf{A} \rhd x \rhd \mathsf{B} \mid \cdots \;\rightarrow\; \langle \mathsf{B}, v\rangle_y \mid \cdots \qquad (y \text{ fresh})$$

Advertisement. Any participant can advertise a new contract C (with preconditions G). This is obtained by performing the step $\Gamma \;\rightarrow\; \Gamma \mid \{G\}C$.

Stipulation. Stipulation turns a contract advertisement into an active contract. For instance, let $G = \mathsf{A}\texttt{:!}\,1\,\texttt{@}\,x \mid \mathsf{A}\texttt{:?}\,1\,\texttt{@}\,y \mid \mathsf{A}\texttt{:secret}\,a$. Given a contract C, the stipulation of $\{G\}C$ is done in a few steps:

$$\langle \mathsf{A}, 1\rangle_x \mid \langle \mathsf{A}, 1\rangle_y \mid \{G\}C \;\rightarrow^*\; \langle \mathsf{A}, 1\rangle_y \mid \langle C, 1\rangle_z \mid \{\mathsf{A} : a\#N\}$$

Above, the funds in the deposit x are transferred to the newly created contract, to fulfill the precondition $\mathsf{A}\texttt{:!}\,1\,\texttt{@}\,x$. Instead, the deposit y remains in the configuration, to be possibly spent after some time. The component $\{\mathsf{A} : a\#N\}$ represents the secret committed to by A, with its length N.

Withdraw. Executing $\texttt{withdraw}\ \mathsf{A}$ terminates the contract, and transfers its whole balance to A by creating a fresh deposit owned by A:

$$\langle \texttt{withdraw}\ \mathsf{A} + C', v\rangle_x \ \rightarrow\ \langle \mathsf{A}, v\rangle_y \qquad (y \text{ fresh})$$

Above, $\texttt{withdraw}\ \mathsf{A}$ is executed as a branch within a choice: as usual, taking a branch discards the other ones (denoted as C').

Split. The $\texttt{split}$ primitive can be used to spawn several new concurrent contracts, dividing the balance among them. For instance:

$$\langle(\texttt{split}\ v_1 \rightarrow C_1 \mid v_2 \rightarrow C_2), v_1 + v_2\rangle_x \rightarrow \langle C_1, v_1\rangle_y \mid \langle C_2, v_2\rangle_z \qquad (y, z \text{ fresh})$$

Reveal. A prefix $\texttt{reveal}\, a\, \texttt{if}\, p$ can be fired when the previously committed secret a (satisfying the predicate p) has been revealed. For instance:

$$\begin{aligned}
&\langle \texttt{reveal}\, a\, \texttt{if}\, a = N.\ C, v\rangle_x \mid \{\mathsf{A} : a\#N\}\\
\rightarrow\ &\langle \texttt{reveal}\, a\, \texttt{if}\, a = N.\ C, v\rangle_x \mid \mathsf{A} : a\#N\\
\rightarrow\ &\langle C, v + v'\rangle_y \mid \mathsf{A} : a\#N
\end{aligned}$$

In the first step, A reveals her secret a. In the second step, any participant fires the prefix.

Authorizations. When a branch is decorated by $\mathsf{A} : \cdots$ it can be taken only after A has provided her authorization. For instance:

$$\begin{aligned}
&\langle \mathsf{A} : \texttt{withdraw}\ \mathsf{B} + \mathsf{A} : \texttt{withdraw}\ \mathsf{C}, v\rangle_x\\
\rightarrow\ &\langle \mathsf{A} : \texttt{withdraw}\ \mathsf{B} + \mathsf{A} : \texttt{withdraw}\ \mathsf{C}, v\rangle_x \mid \mathsf{A} \rhd x \rhd \mathsf{A} : \texttt{withdraw}\ \mathsf{B}\\
\rightarrow\ &\langle \mathsf{B}, v\rangle_y
\end{aligned}$$

In the first step, A authorizes to take the branch $\texttt{withdraw}\ \mathsf{B}$. After that, any participant can fire such branch.

Time. We always allow time t to advance by a delay $\delta > 0$, through a transition $\Gamma \mid t \rightarrow \Gamma \mid t + \delta$. Advancing time can enable branches decorated with $\texttt{after}\, t$. For instance, if $t_0 + \delta \geq t$, we have the following computation:

$$\begin{aligned}
&\langle(\texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}) + C', v\rangle_x \mid t_0\\
\rightarrow\ &\langle(\texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}) + C', v\rangle_x \mid t_0 + \delta \ \rightarrow\ \langle \mathsf{B}, v\rangle_y \mid t_0 + \delta
\end{aligned}$$

2.2.1 Runs and strategies.

A *run* $\mathcal{S}$ is a (possibly infinite) sequence:

$$\Gamma_0 \mid t_0 \xrightarrow{\alpha_0} \Gamma_1 \mid t_1 \xrightarrow{\alpha_1} \cdots$$

where α_i are the transition labels, Γ_0 contains only deposits, and $t_0 = 0$. If $\mathcal{S}$ is finite, we write $\Gamma_{\mathcal{S}}$ for its last untimed configuration, and $\delta_{\mathcal{S}}$ for its last time. A *strategy* Σ^s_{A} is a PPTIME algorithm which allows A to select which actions to perform (possibly, time delays), among those permitted by the BitML semantics. The choice among these actions is controlled by the adversary strategy Σ^s_{Adv}, which acts on behalf of all the dishonest participants. Given the strategies of all participants (including Adv), there is a unique run *conforming* to all of them.

2.3 Liquidity

A desirable property of smart contracts is *liquidity*, which requires that the contract balance is always eventually transferred to some participant. In a non-liquid contract, funds can be frozen forever, unavailable to anyone, hence effectively destroyed. There are many possible flavours of liquidity, depending e.g. on which participants are assumed to be honest, and on which are their strategies. The simplest form of liquidity is to consider the case where everyone cooperates: i.e. a contract is liquid if there exists some strategy for each participant such that no funds are ever frozen. However, this notion does not capture the essence of smart contracts, i.e. to allow mutually untrusted participants to safely interact.

For instance, consider the following contract, where A and B contribute 1Ƀ each for a donation of 2Ƀ to either C or D (we omit the preconditions for brevity):

$$\mathsf{A} : \mathsf{B} : \texttt{withdraw}\ \mathsf{C} \ + \ \mathsf{A} : \mathsf{B} : \texttt{withdraw}\ \mathsf{D}$$

In order to unlock the funds, A and B must agree on the recipient of the donation, by giving their authorization on the same branch. This contract would be liquid only by assuming the cooperation between A and B: indeed, A alone cannot guarantee that the 2Ƀ will eventually be donated, as B can choose a different recipient, or even refuse to give any authorization. Consequently, unless A trusts B, it makes sense to consider this contract as non-liquid, from the point of view of A (and for similar reasons, also from that of B).

Consider now the timed commitment contract discussed before:

$$\texttt{reveal}\, a.\, \texttt{withdraw}\ \mathsf{A} \ + \ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}$$

This contract is liquid from A's point of view (even if B is dishonest), because A can reveal the secret and then redeem the funds from the contract. The timed commitment is also liquid from B's point of view: if A does not reveal the secret (making the first branch stuck), the funds in the contract can be redeemed through the second branch, after time t.

In a *mutual* timed commitment contract, where A and B have to exchange their secrets or pay a 1₿ penalty, achieving liquidity is a bit more challenging. We first consider a wrong attempt:

$$\begin{aligned}&\ \ \texttt{reveal}\, a.\,\texttt{reveal}\, b.\,\texttt{split}\ (1₿ \to \texttt{withdraw}\ \mathsf{A} \mid 1₿ \to \texttt{withdraw}\ \mathsf{B})\\ &+ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\end{aligned}$$

Intuitively, A has only the following strategies, according to when she decides to reveal her secret a: (i) A chooses to reveal a unconditionally, and to perform the `reveal` a action. This strategy is *not* liquid: indeed, if B does not reveal b, the contract is stuck. (ii) A chooses to reveal a only *after* B has revealed b. This strategy is *not* liquid: indeed, if B chooses not to reveal b, the contract will never advance. (iii) A chooses to wait until B reveals secret b, or until time $t' \geq t$, whichever comes first. If b was revealed, A reveals a, and splits the contract balance between A and B. Otherwise, if the deadline t' is expired, A transfers the whole balance to B. Note that, although this strategy is liquid, it is not satisfactory for A, since in the second case she will lose money.

This example highlights a crucial point: participants' strategies have to be taken into account when defining liquidity. Indeed, the mere fact that a liquid strategy exists does not imply that it is the ideal strategy for the honest participant. To fix this issue, we revise the mutual timed commitment as follows:

$$\begin{aligned}&\ \ \texttt{reveal}\, a.\,\big(\texttt{reveal}\, b.\,\texttt{split}\ (1₿ \to \texttt{withdraw}\ \mathsf{A} \mid 1₿ \to \texttt{withdraw}\ \mathsf{B})\\ &\qquad\qquad + \texttt{after}\, t' : \texttt{withdraw}\ \mathsf{A}\big)\\ &+ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\end{aligned}$$

where $t < t'$. Now, A has a liquid strategy where she does not pay the penalty. First, A reveals a before time t. After that, if B reveals b, then A can execute the `split`, transferring 1₿ to herself and 1₿ to B (note that this does not require B's cooperation); otherwise, after time t', A can withdraw 2₿ by executing the `withdraw` A in the `after` $t' : \cdots$ branch.

These examples, albeit elementary, show that detecting if a strategy is liquid for a contract is not straightforward, in general. The problem of determining a liquid strategy for a given contract seems even more demanding. Automatic techniques for the verification and inference of liquid strategies can be useful tools for the developers of smart contracts.

2.3.1 Verifying liquidity

One of the main features of BitML is a verification technique for the liquidity of BitML contracts. The technique is based on a more general result, i.e. a strict correspondence between the concrete semantics of BitML and a new abstract semantics, which is finite-state. The abstraction is a correct and complete approximation of the concrete semantics with respect to a given set of contracts [24]. To obtain a finite-state abstraction, we need to cope with three sources of infiniteness of the concrete semantics of BitML: the unbounded passing of time, the advertisement/stipulation of new contracts, and the operations on deposits. The abstraction replaces the time t in concrete configurations with a finite number of time intervals $T = [t_0, t_1)$, and it disables the transitions to advertise new contracts. Further, the only operations on deposits allowed by the abstract semantics are the ones for transferring them to contracts and for destroying them. The latter is needed e.g. to properly model the situation where a participant spends a ?-deposit.

The intended use of our abstraction is to start from a configuration containing an arbitrary (but finite) set of contracts, and then analyse their possible evolutions in the presence of an honest participant and an adversary. This produces a finite set of (finite) traces, which we can model-check for liquidity. Soundness and completeness of the abstraction are exploited to prove that liquidity is decidable. The computational soundness of the BitML compiler [22] guarantees that if a contract is verified to be liquid according to our analysis, this property is preserved when executing it on Bitcoin.

Part II

Contributions

Chapter 3

A Formal Model of Bitcoin Transactions

Bitcoin [116], the first decentralized cryptocurrency, was introduced in 2009, and through the years it has consolidated its position as the most popular one. Bitcoin and other cryptocurrencies have pushed forward the concept of decentralization, providing means for reliable interactions between mutually distrusting parties on an open network.

Besides the intended monetary application, the Bitcoin blockchain can be seen as a way to consistently maintain the state of a system over a peer-to-peer network, without the need of a trusted authority. If the system is a currency, its state is the amount of funds in each account. This concept can be generalised to the case where the system is a *smart contract* [72], namely an executable computer protocol which can also handle transfers of currency. The idea of exploiting the Bitcoin blockchain to build smart contracts has recently been explored by several works. Lotteries [4, 26, 62, 23], gambling games [57], contingent payments [12], covenants [63, 66], and other kinds of fair computations [1, 56] are some examples of the capabilities of Bitcoin as a platform for smart contracts.

Smart contracts often rely on features of Bitcoin that go beyond the standard transfers of currency. For instance, while the vast majority of Bitcoin transactions uses scripts only to verify signatures, smart contracts like the above-mentioned ones exploit more complex scripts, e.g. to determine the winner of a lottery, or to check if a secret has been revealed. Smart contracts may also exploit other (infrequently used) features of Bitcoin, e.g. various signature modifiers, and temporal constraints on transactions.

As a matter of fact, using these advanced features to design a new smart contract is not a trivial matter, for two reasons. First, while the overall behaviour of Bitcoin is clear, the details of many of its crucial aspects are poorly documented. To understand the details of how a mechanism actually works, one has to explore various web pages (often inaccurate, or inconsistent, or overly technical), and eventually resort to the source code of the Bitcoin client [83] to have the correct answer. Second, the description of advanced features is often too concrete to be effectively used in the design and analysis of a smart contract (indeed, in many cases the only available description coincides with the implementation).

This Chapter is structured as follows. In Section 3.1 we formalise Bitcoin transactions. Besides transactions, we also provide an high-level model of the blockchain, and we study its basic properties. In Section 3.2 we illustrate, through a basic case study, the impact of the Segregated Witness feature on the expressiveness of Bitcoin smart contracts. In Section 3.3 we show how to translate transactions from our model to standard Bitcoin transactions. We discuss the differences between our model and the actual Bitcoin in Section 7.3.

3.1 The model

In this section we propose a formal model of Bitcoin transactions, which is sufficiently abstract to enable formal reasoning, and at the same time is concrete enough to serve as an alternative documentation to Bitcoin. We use our model to formally prove some well-formedness properties of the Bitcoin blockchain, for instance that each transaction can only be spent once.

In Section 3.1.1 we define the scripts that can be used in transaction outputs. Then, in Section 3.1.2 we formalise transactions, and in Section 3.1.3 we define a signature scheme for them. Sections 3.1.4 and 3.1.5 give semantics, respectively, to scripts and transactions. In Section 3.1.6 we model the Bitcoin blockchain, and in particular we define the crucial notion of *consistency*, which corresponds to the one enforced by the Bitcoin consensus protocol. We then state a few results about consistent blockchains.

We start by introducing some auxiliary notation. We assume several sets, ranged over by meta-variables as shown in the left column of Table 3.1. We use the bold notation to denote finite sequences of elements. We denote with $\boldsymbol{x}_i$ the i-th element of a sequence $\boldsymbol{x}$, i.e. $\boldsymbol{x}_i = x_i$ if $\boldsymbol{x} = x_1 \dots x_n$, and with $\boldsymbol{x}_{i..j}$ the subsequence of $\boldsymbol{x}$ starting from the i-th element and ending to the j-th element. We denote with $|\boldsymbol{x}|$ the num-

$\mathsf{A}, \mathsf{B}, \ldots \in \mathsf{Part}$	Participants	$e, e', \ldots \in \mathsf{Exp}$	Script expressions
$x, y, \ldots \in \mathsf{Var}$	Variables	$\mathsf{T}, \mathsf{T}', \ldots \in \mathsf{Tx}$	Transactions
$\nu, \nu', \ldots \in \mathsf{Den}$	Denotations, i.e.:	μ, μ'	Signature modifier
$k, k' \ldots \in \mathbb{Z}$	Constants	$\mathsf{sig}_k^{\mu,i}(\mathsf{T})$	Transaction signature
$t, t' \ldots \in \mathbb{N}$	Time	$\mathsf{ver}_k(\sigma, \mathsf{T}, i)$	Signature verification
$v, v' \ldots \in \mathbb{N}$	Currency values	$\mathsf{T}, i \models \lambda \boldsymbol{x}.e$	Script verification
$\sigma, \sigma', \ldots \in \mathbb{Z}$	Signatures	$(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$	Transaction redeem
true, *false*	Boolean values	$\mathbf{B} = (\mathsf{T}_1, t_1) \cdots$	Blockchains
$\perp$	Undefined	$\mathbf{B} \triangleright (\mathsf{T}, t)$	Consistent update

Table 3.1: *Summary of notation.*

ber of elements of $\boldsymbol{x}$, and with ε the empty sequence. We denote with $f : A \rightharpoonup B$ a *partial* function f from A to B, with $\operatorname{dom} f$ the *domain* of f, i.e. the subset of A where f is defined, and with $\operatorname{ran} f$ the *range* of f, i.e. $\operatorname{ran} f = \{f(x) \mid x \in \operatorname{dom} f\}$. We use $\perp$ to represent an "undefined" element; in particular, when the element is a partial function, $\perp$ denotes the function with empty domain. For a pair (x, y), we define $\mathit{fst}(x, y) = x$ and $\mathit{snd}(x, y) = y$.

3.1.1 Scripts

Each output in a Bitcoin transaction contains a script, which is used to establish when the output can be redeemed by another transaction. Intuitively, a script is a first-order function (written in a non Turing-equivalent language), which is applied to the witness provided by the redeeming transaction. The output can be redeemed only if such function application evaluates to true.

In our model, we abstract from the actual stack-based scripting language implemented in Bitcoin [89], by using instead a minimalistic language of expressions.

Definition 3.1 (Scripts)**.** *We define the set* Exp *of* script expressions *(ranged over by $e, e', \ldots$) as follows:*

$$e ::= x \mid k \mid e + e \mid e - e \mid e = e \mid e < e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid |e| \mid \mathsf{H}(e) \mid \mathsf{versig}(\boldsymbol{k}, \boldsymbol{e}) \mid \mathsf{absAfter}\ t : e \mid \mathsf{relAfter}\ t : e$$

We denote with Script *the set of terms of the form $\lambda \boldsymbol{z}.e$ such that all the variables in e occur in $\boldsymbol{z}$.*

Besides some basic arithmetic and logical operators, script expressions include a few operators inspired from the actual Bitcoin scripting language. The expression $|e|$ denotes the size, in bytes, of the evaluation of e. The expression $\mathsf{H}(e)$ evaluates to the hash of e. The expression $\mathsf{versig}(\boldsymbol{k}, \boldsymbol{e})$ takes as arguments a sequence of m script expressions, representing signatures of the enclosing transactions, and a sequence of n public keys. Intuitively, it evaluates to true whenever the provided signatures are verified by using m out of the n provided keys. The expressions $\mathsf{absAfter}\ t : e$ and $\mathsf{relAfter}\ t : e$ define temporal constraints (see Section 3.1.4). They evaluate to e if the constraints are satisfied, otherwise they fail.

Notation 3.2. *We use the following syntactic sugar for expressions: (i) false to denote* $1 = 0$ *(ii) true to denote* $1 = 1$ *(iii)* $e \wedge e'$ *to denote* $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ \mathit{false}$ *(iv)* $e \vee e'$ *to denote* $\mathsf{if}\ e\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ e'$ *(v)* $\mathsf{not}\ e$ *to denote* $\mathsf{if}\ e\ \mathsf{then}\ \mathit{false}\ \mathsf{else}\ \mathit{true}$*.*

3.1.2 Transactions

The following definition formalises Bitcoin transactions.

Definition 3.3 (Transactions)**.** *We inductively define the set* Tx *of transactions as follows. A transaction* T *is a tuple* $(\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock})$*, where:*

- $\mathsf{in} : \mathbb{N} \rightharpoonup \mathsf{Tx} \times \mathbb{N}$
- $\mathsf{wit} : \mathbb{N} \rightharpoonup \mathbb{Z}^*$*, where* $\mathrm{dom}\,\mathsf{wit} = \mathrm{dom}\,\mathsf{in}$
- $\mathsf{out} : \mathbb{N} \rightharpoonup \mathsf{Script} \times \mathbb{N}$
- $\mathsf{absLock} : \mathbb{N}$
- $\mathsf{relLock} : \mathbb{N} \rightharpoonup \mathbb{N}$*, where* $\mathrm{dom}\,\mathsf{relLock} = \mathrm{dom}\,\mathsf{in}$

where, for all $i, j \in \mathrm{dom}\,\mathsf{in}$*,* $\mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$ *and* $i \neq j \implies \mathsf{in}(i) \neq \mathsf{in}(j)$*.*

We denote with $\mathsf{T.f}$ *the value of field* f *of* T*, for* $\mathsf{f} \in \{\mathsf{in}, \mathsf{wit}, \mathsf{out}, \mathsf{absLock}, \mathsf{relLock}\}$*.*

We say that T *is* initial *when* $\mathsf{T.in} = \mathsf{T.relLock} = \bot$ *and* $\mathsf{T.absLock} = 0$*.*

The fields in and out represent, respectively, the inputs and the outputs of a transaction. There is an input for each $i \in \mathrm{dom}\,\mathsf{in}$, and an output for each $j \in \mathrm{dom}\,\mathsf{out}$. When $\mathsf{T.in}(i) = (\mathsf{T}', j)$, it means that the i-th input of T wants to redeem the j-th output of T'. The side condition $i \neq j \Rightarrow \mathsf{in}(i) \neq \mathsf{in}(j)$ ensures that inputs are pairwise distinct. The side condition $\mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$ is related to the *Segregated Witness* (SegWit) feature and it requires that the witness of the input transaction is left unspecified. This feature, specified in the BIP 141 [113] and activated on August 24th 2017, implies that witnesses are not used in the computation of transaction hashes. The output $\mathsf{T}'.\mathsf{out}(j)$ is a pair $(\lambda \boldsymbol{z}.e, v)$, meaning that v Satoshis ($1\text{₿} = 10^8$ Satoshis) can be redeemed by whoever can provide a witness which makes the term $\lambda \boldsymbol{z}.e$ evaluate to *true*. Such witness is defined by $\mathsf{T}.\mathsf{wit}(i)$. The fields $\mathsf{T}.\mathsf{absLock}$ and $\mathsf{T}.\mathsf{relLock}(i)$ specify a constraint on when T can be put on the blockchain: the first in absolute terms, whereas the second is relative to the transaction in the input $\mathsf{T}.\mathsf{in}(i)$. More specifically, $\mathsf{T}.\mathsf{absLock} = t$ means that T can appear on the blockchain only after time t. If $\mathsf{T}.\mathsf{relLock}(i) = t$, then T can appear only after time t since the transaction in $\mathsf{T}.\mathsf{in}(i)$ appeared.

To improve readability, we use the following conventions: (i) if T has exactly one input, we denote it by $\mathsf{T}.\mathsf{in}$ (omitting the index, which we assume to be 1); We act similarly for $\mathsf{T}.\mathsf{wit}$, $\mathsf{T}.\mathsf{out}$, and $\mathsf{T}.\mathsf{relLock}$; (ii) if $\mathsf{T}.\mathsf{absLock} = 0$, we omit it (similarly for $\mathsf{T}.\mathsf{relLock}$ when it is $\bot$); (iii) we denote with $\mathsf{scr}(\mathsf{T}.\mathsf{out}(i))$ and $\mathsf{val}(\mathsf{T}.\mathsf{out}(i))$, respectively, the first and the second element of the pair $\mathsf{T}.\mathsf{out}(i)$.

3.1.3 Transaction signatures

We extend to transactions the signing and verification functions of the public key signature schemes, denoted respectively as $\mathit{sig}_k(\cdot)$ and $\mathit{ver}_k(\cdot,\cdot)$. For simplicity, although we will always use $k = (k_p, k_s)$ for key pairs, we implicitly assume that $\mathit{sig}_k(\cdot)$ only uses the private part k_s, while $\mathit{ver}_k(\cdot,\cdot)$ only uses the public part k_p.

In Bitcoin, transaction signatures never apply to the whole transaction: users can specify which parts of a transaction are signed (with the exception of the wit field, which is never signed). However, not all possible combinations of transaction parts are possible; the legit ones are listed in Definition 3.5. In order to specify which parts of a transaction are signed, we first introduce the auxiliary notion of *transaction substitution.*

Definition 3.4 (Transaction substitutions)**.** *A transaction substitution Σ is a function from Tx to Tx. For a transaction field f, we denote with $\{\mathsf{f} \mapsto d\}$ the substitution which replaces the value of f with d. For $\mathsf{f} \neq \mathsf{absLock}$ and $i \in \mathbb{N}$, we denote with $\{\mathsf{f}(i) \mapsto d\}$ the substitution which replaces $\mathsf{f}(i)$ with d. Further, for $\circ \in \{<, >, \neq\}$, we denote with $\{\mathsf{f}(\circ\, i) \mapsto$*

$$\begin{aligned}
**_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit}(1) \mapsto i\}\{\mathsf{wit}(\neq 1) \mapsto \bot\} \\
*0_i(\mathsf{T}) &= **_i(\mathsf{T}\{\mathsf{out} \mapsto \bot\}) \\
*1_i(\mathsf{T}) &= **_i(\mathsf{T}\{\mathsf{out}(< i) \mapsto (\mathit{false}, 0)\}\{\mathsf{out}(> i) \mapsto \bot\}) \\
1*_i(\mathsf{T}) &= **_1(\mathsf{T}\{\mathsf{in}(1) \mapsto \mathsf{T}.\mathsf{in}(i)\}\{\mathsf{in}(\neq 1) \mapsto \bot\} \\
&\qquad \{\mathsf{relLock}(1) \mapsto \mathsf{T}.\mathsf{relLock}(i)\}\{\mathsf{relLock}(\neq 1) \mapsto \bot\}) \\
10_i(\mathsf{T}) &= 1*_i(*0_i(\mathsf{T})) \\
11_i(\mathsf{T}) &= 1*_i(*1_i(\mathsf{T}))
\end{aligned}$$

Figure 3.1: *Signature modifiers.*

$d\}$ the substitution which replaces $\mathsf{f}(j)$ with d, for all $j \circ i \in \operatorname{dom} \mathsf{f}$.

Definition 3.5 (Signature modifiers)**.** *We define signature modifiers $\mu[i]$ (with $i \in \mathbb{N}$) in Figure 3.1. We associate to each modifier a substitution, and we denote with $\mu[i](\mathsf{T})$ the result of applying it to the transaction T.*

Each modifier is represented by a pair of symbols, describing, respectively, the set of inputs and of outputs being signed ($*$ = all, 1 = single, 0 = none), and an index $i \in \mathbb{N}$. The index has different meanings, depending on the modifier. Regarding the first symbol of the modifier, if it is $*$, then i is the index of the witness where the signature will be included, so to ensure that a signature computed for being included in the witness at index i can not be used in any witness with index $j \neq i$ (see Example 3.1.4). If the first symbol of the modifier is 1, then only the i-th input is signed, while all the other inputs are removed from the transaction. With respect to the second symbol of the modifier, if it is 1, then i is the index of the signed output; otherwise, i has no effect on the outputs to be signed. Note that a single index is used for both inputs and outputs: in any case, the index refers to the witness where the signature will be included.

Definition 3.6 (Transaction signatures)**.** *We define the transaction signature (under modifier μ and index i) and verification functions as follows:*

$$\begin{aligned}
\mathsf{sig}_k^{\mu,i}(\mathsf{T}) &= (\mathit{sig}_k(\mu[i](\mathsf{T}), \mu), \mu) \\
\mathsf{ver}_k(\sigma, \mathsf{T}, i) &= \mathit{ver}_k(w, (\mu[i](\mathsf{T}), \mu)) \quad \text{if } \sigma = (w, \mu)
\end{aligned}$$

Hereafter, we use $\sigma, \sigma', \ldots$ to range over transaction signatures.

Note that a signature $\sigma = (sig_k((\mu[i](\mathsf{T}), \mu)), \mu)$ does not contain the index i. Consequently, the verification function requires i to be passed as parameter, i.e. we write $\mathsf{ver}_k(\sigma, \mathsf{T}, i)$. The parameter i will be instantiated by the script verification function (see Definition 3.10). Besides the modified transaction $\mu[i](\mathsf{T})$, the signature also applies to the modifier μ. In this way, signing a single-input transaction T with modifier $**_1$ and with modifier $1*_1$ results in two different signatures, even though $**_1(\mathsf{T}) = 1*_1(\mathsf{T})$.

Notation 3.7. *Note that* $\mathsf{sig}_k^{\mu,i}(\mathsf{T})$ *can meaningfully appear within* $\mathsf{T}.\mathsf{wit}(i)$*, since such signature does not depend on the* wit *field of transactions (as all signature modifiers overwrite all the witnesses). When a signature of* T *appears within* $\mathsf{T}.\mathsf{wit}(i)$*, as a shorthand we denote it with* sig_k^{μ} *(so, neglecting the enclosing transaction* T *and the index i), or just* sig_k *when* $\mu = **$*.*

We now extend the signature verification $\mathsf{ver}_k(\sigma, \mathsf{T}, i)$ to the case where, instead of providing a single key k and a single signature σ, one has many keys and signatures, i.e. $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$. Intuitively, if $|\boldsymbol{\sigma}| = m$ and $|\boldsymbol{k}| = n$, the function $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i)$ implements a m-of-n multi-signature scheme, i.e. it evaluates to true if all the m signatures match (some of) the keys in $\boldsymbol{k}$. The actual definition is a bit more complex, to be coherent with the one implemented in Bitcoin.

Definition 3.8 (Multi-signature verification)**.** *Let* $\boldsymbol{k}$ *and* $\boldsymbol{\sigma}$ *be sequences of (public) keys and signatures such that* $|\boldsymbol{k}| \geq |\boldsymbol{\sigma}|$*, and let* $i \in \mathbb{N}$*. For all* $m, n \in \mathbb{N}$*, we define the function:*

$$\mathsf{ver}_{\boldsymbol{k}}^{n,m}(\boldsymbol{\sigma}, \mathsf{T}, i) \equiv \begin{cases} \mathit{true} & \mathit{if}\ m = 0 \\ \mathit{false} & \mathit{if}\ m \neq 0\ \mathit{and}\ n = 0 \\ \mathsf{ver}_{\boldsymbol{k}}^{n-1,m-1}(\boldsymbol{\sigma}, \mathsf{T}, i) & \mathit{if}\ m, n \neq 0\ \mathit{and}\ \mathsf{ver}_{\boldsymbol{k}_n}(\boldsymbol{\sigma}_m, \mathsf{T}, i) \\ \mathsf{ver}_{\boldsymbol{k}}^{n-1,m}(\boldsymbol{\sigma}, \mathsf{T}, i) & \mathit{otherwise} \end{cases}$$

Then, we define $\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, i) = \mathsf{ver}_{\boldsymbol{k}}^{|\boldsymbol{k}|,|\boldsymbol{\sigma}|}(\boldsymbol{\sigma}, \mathsf{T}, i)$*.*

Our formalisation of multi-signature verification (Definition 3.8) follows closely the implementation of Bitcoin, whose stack-based scripting language imposes that the sequence $\boldsymbol{\sigma}$ is read in reverse order. Accordingly, the function ver tries to verify the last signature in $\boldsymbol{\sigma}$ with the last key in $\boldsymbol{k}$. If they match, the function ver proceeds to verify the previous signature in the sequence, otherwise it tries to verify the signature with the previous key.

Example 3.1.1 (2-of-3 multi-signature)**.** Let $\boldsymbol{k} = k_a k_b k_c$, and let $\boldsymbol{\sigma} = \sigma_p \sigma_q$ be such that $\mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathsf{ver}_{k_b}(\sigma_q, \mathsf{T}, 1) = \mathit{true}$, and false otherwise. We have that:

$$\begin{aligned}
\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma}, \mathsf{T}, 1) &= \mathsf{ver}_{\boldsymbol{k}}^{3,2}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } |\boldsymbol{k}| = 3 \text{ and } |\boldsymbol{\sigma}| = 2)\\
&= \mathsf{ver}_{\boldsymbol{k}}^{2,2}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_c}(\sigma_q, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}_{\boldsymbol{k}}^{1,1}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_b}(\sigma_q, \mathsf{T}, 1) = \mathit{true})\\
&= \mathsf{ver}_{\boldsymbol{k}}^{0,0}(\boldsymbol{\sigma}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathit{true})\\
&= \mathit{true} && (\text{as } m = 0)
\end{aligned}$$

Note that, if we let $\boldsymbol{\sigma'} = \sigma_q \sigma_p$, the resulting evaluation will be:

$$\begin{aligned}
\mathsf{ver}_{\boldsymbol{k}}(\boldsymbol{\sigma'}, \mathsf{T}, 1) &= \mathsf{ver}_{\boldsymbol{k}}^{3,2}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } |\boldsymbol{k}| = 3 \text{ and } |\boldsymbol{\sigma'}| = 2)\\
&= \mathsf{ver}_{\boldsymbol{k}}^{2,2}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_c}(\sigma_p, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}_{\boldsymbol{k}}^{1,2}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_b}(\sigma_p, \mathsf{T}, 1) = \mathit{false})\\
&= \mathsf{ver}_{\boldsymbol{k}}^{0,1}(\boldsymbol{\sigma'}, \mathsf{T}, 1) && (\text{as } \mathsf{ver}_{k_a}(\sigma_p, \mathsf{T}, 1) = \mathit{true})\\
&= \mathit{false} && (\text{as } m \neq 0 \text{ and } n = 0)
\end{aligned}$$

□

3.1.4 Semantics of scripts

Definition 3.9 gives the semantics of script expressions. This semantics will be used in Section 3.1.5 to define when a transaction can redeem another one. We use an environment $\rho : \mathsf{Var} \rightharpoonup \mathbb{Z}$ which associates a denotation to each variable occurring in it. Further, we use a transaction $\mathsf{T} \in \mathsf{Tx}$ and an index $i \in \mathbb{N}$ to indicate the witness redeeming the script, both used to evaluate the timelock expressions. We use the denotation $\perp$ to represent "failure" of the evaluation. This is the case e.g. of timelock expressions, when the temporal constraint is not satisfied. All the semantic operators used in Definition 3.9 are *strict*, i.e. they evaluate to $\perp$ if some of their operands is $\perp$.

Definition 3.9 (Expression evaluation)**.** *Let* $\rho : \mathsf{Var} \rightharpoonup \mathbb{Z}$, *let* $\mathsf{T} \in \mathsf{Tx}$ *and* $i \in \mathbb{N}$. *We define the function* $[\![\cdot]\!]_{\mathsf{T},i,\rho} : \mathsf{Exp} \to \mathsf{Den}$ *in Figure 3.2, where*

$$
\begin{aligned}
[\![x]\!]_{\mathsf{T},i,\rho} &= \rho(x)\\
[\![k]\!]_{\mathsf{T},i,\rho} &= k\\
[\![\mathsf{versig}(\boldsymbol{k},\boldsymbol{e})]\!]_{\mathsf{T},i,\rho} &= \mathsf{ver}_{\boldsymbol{k}}([\![\boldsymbol{e}]\!]_{\mathsf{T},i,\rho},\mathsf{T},i)\\
[\![\mathsf{H}(e)]\!]_{\mathsf{T},i,\rho} &= H([\![e]\!]_{\mathsf{T},i,\rho}) \qquad (H \text{ is a public hash function})\\
[\![\mathsf{absAfter}\ t : e]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ \mathsf{T}.\mathsf{absLock} \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot\\
[\![\mathsf{relAfter}\ t : e]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ \mathsf{T}.\mathsf{relLock}(i) \geq t\ \mathit{then}\ [\![e]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ \bot\\
[\![e \circ e']\!]_{\mathsf{T},i,\rho} &= [\![e]\!]_{\mathsf{T},i,\rho} \circ_\bot [\![e']\!]_{\mathsf{T},i,\rho} \qquad (\circ \in \{+,-,=,<\})\\
[\![|e|]\!]_{\mathsf{T},i,\rho} &= \mathit{size}([\![e]\!]_{\mathsf{T},i,\rho})\\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_{\mathsf{T},i,\rho} &= \mathit{if}\ [\![e_0]\!]_{\mathsf{T},i,\rho}\ \mathit{then}\ [\![e_1]\!]_{\mathsf{T},i,\rho}\ \mathit{else}\ [\![e_2]\!]_{\mathsf{T},i,\rho}
\end{aligned}
$$

Figure 3.2: *Semantics of script expressions.*

$\mathsf{Den} = \mathbb{Z} \cup \{\mathit{true}, \mathit{false}\}$. *We use the following operators on denotations:*

$$
\mathit{if}\ \nu_0\ \mathit{then}\ \nu_1\ \mathit{else}\ \nu_2 \equiv \begin{cases} \nu_1 & \mathit{if}\ \nu_0 = \mathit{true}\\ \nu_2 & \mathit{if}\ \nu_0 = \mathit{false}\\ \bot & \mathit{otherwise} \end{cases}
$$

$$
\mathit{size}(\nu) \equiv \begin{cases} \bot & \mathit{if}\ \nu \notin \mathbb{Z}\\ 0 & \mathit{if}\ \nu = 0\\ \left\lceil \frac{\log_2 |\nu|+1}{7} \right\rceil & \mathit{otherwise} \end{cases}
$$

$$
\begin{aligned}
\nu_0 \circ_\bot \nu_1 \equiv\ & \mathit{if}\ \nu_0,\nu_1 \in \mathbb{Z}\ \mathit{then}\ \nu_0 \circ \nu_1\ \mathit{else}\ \bot\\
& (\circ \in \{+,-,=,<\})
\end{aligned}
$$

The function $\mathit{size}(\nu)$ denotes the amount of bytes that are necessary to represent the value ν, as defined in the Bitcoin scripting language [89].

Definition 3.10 (Script verification)**.** *We say that the input i of* T verifies *$\lambda\boldsymbol{x}.e$ (in symbols: $\mathsf{T}, i \models \lambda\boldsymbol{x}.e$) when $\boldsymbol{x} = x_1 \ldots x_n$, $\mathsf{T}.\mathsf{wit}(i) = k_1 \ldots k_n$, and:*

$$
[\![e]\!]_{\mathsf{T},i,\{x_j \mapsto k_j \,|\, j \in 1\ldots n\}} = \mathit{true}
$$

Example 3.1.2. Let H be a hash function, let $s, h \in \mathbb{Z}$ be such that $h = H(s)$, and let T be such that $\mathsf{T}.\mathsf{wit}(1) = (\sigma, s)$, with $\sigma = \mathsf{sig}_k^{**}(\mathsf{T})$.

We prove that:

$$\mathsf{T}, 1 \models \lambda(\varsigma, x).\big(\mathsf{versig}(k, \varsigma) \text{ and } \mathsf{H}(x) = h\big)$$

To do this, let $\rho = \{\varsigma \mapsto \sigma, x \mapsto s\}$. We have that:

$$\begin{aligned}
[\![\mathsf{versig}(k,\varsigma) \text{ and } \mathsf{H}(x) = h]\!]_{\mathsf{T},1,\rho} &= [\![\mathsf{versig}(k,\varsigma)]\!]_{\mathsf{T},1,\rho} \ \mathit{and}\ [\![\mathsf{H}(x) = h]\!]_{\mathsf{T},1,\rho} \\
&= \mathsf{ver}_k([\![\varsigma]\!]_{\mathsf{T},1,\rho}, \mathsf{T}, 1) \ \mathit{and}\ ([\![\mathsf{H}(x)]\!]_{\mathsf{T},1,\rho} =_\perp [\![h]\!]_{\mathsf{T},1,\rho}) \\
&= \mathsf{ver}_k(\rho(\varsigma), \mathsf{T}, 1) \ \mathit{and}\ (H([\![x]\!]_{\mathsf{T},1,\rho}) =_\perp h) \\
&= \mathsf{ver}_k(\sigma, \mathsf{T}, 1) \ \mathit{and}\ (H(\rho(x)) =_\perp h) \\
&= \mathit{true}
\end{aligned}$$

□

3.1.5 Semantics of transactions

Definition 3.11 describes when the j-th input of a transaction T' (put on the blockchain at time t') can redeem v Satoshis from the i-th output of the transaction T (put on the blockchain at time t). We denote this by $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Definition 3.11 (Output redeeming)**.** *We write* $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$ *iff all the following conditions hold:*

(a) $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{T}\{\mathsf{wit} \mapsto \perp\}, i)$

(b) $\mathsf{T}', j \models \mathsf{scr}(\mathsf{T}.\mathsf{out}(i))$

(c) $v = \mathsf{val}(\mathsf{T}.\mathsf{out}(i))$

(d) $t' \geq \mathsf{T}'.\mathsf{absLock}$

(e) $t' - t \geq \mathsf{T}'.\mathsf{relLock}(j)$

We write $(\mathsf{T}, i, t) \not\rightsquigarrow (\mathsf{T}', j, t')$ *when for no* v *it holds that* $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Item (a) links the j-th input of T' to the i-th output of T. Note that, since we are modelling Segregated Witness, the witness in the transaction $\mathsf{T}'.\mathsf{in}(j)$ is left unspecified: this is why we set to $\perp$ also the witness of T. Item (b) requires that the j-th witness of T' verifies the i-th output script of T. Item (c) just defines v as the value in the i-th output of T. Items (d) and (e) check the absolute and relative timelocks, respectively. The first constraint states that T' cannot appear on the blockchain before $\mathsf{T}'.\mathsf{absLock}$; the second one states that T' cannot appear until at least $\mathsf{T}'.\mathsf{relLock}(j)$ time units have elapsed since T was put on the blockchain.

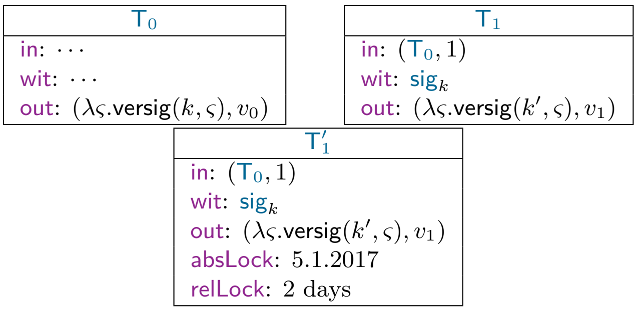

T_0
in: $\cdots$
wit: $\cdots$
out: $(\lambda\varsigma.\mathsf{versig}(k,\varsigma), v_0)$
T_1
in: $(\mathsf{T}_0, 1)$
wit: sig_k
out: $(\lambda\varsigma.\mathsf{versig}(k',\varsigma), v_1)$
T'_1
in: $(\mathsf{T}_0, 1)$
wit: sig_k
out: $(\lambda\varsigma.\mathsf{versig}(k',\varsigma), v_1)$
$\mathsf{absLock}$: 5.1.2017
$\mathsf{relLock}$: 2 days

Figure 3.3: *Three transactions. For notational conciseness, when displaying transactions we omit the substitution* $\{\mathsf{wit} \mapsto \perp\}$ *for the transaction within the* in *field (e.g., we just write* T_0 *within* $\mathsf{T}_1.\mathsf{in}$*). Also, we use dates in time constraints.*

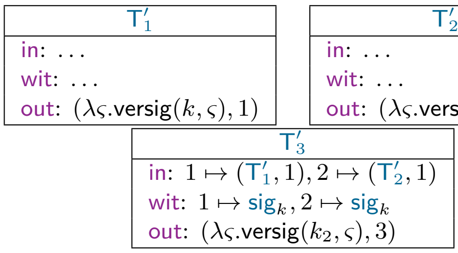

T'_1
in: $\ldots$
wit: $\ldots$
out: $(\lambda\varsigma.\mathsf{versig}(k,\varsigma), 1)$
T'_2
in: $\ldots$
wit: $\ldots$
out: $(\lambda\varsigma.\mathsf{versig}(k,\varsigma), 2)$
T'_3
in: $1 \mapsto (\mathsf{T}'_1, 1), 2 \mapsto (\mathsf{T}'_2, 1)$
wit: $1 \mapsto \mathsf{sig}_k, 2 \mapsto \mathsf{sig}_k$
out: $(\lambda\varsigma.\mathsf{versig}(k_2,\varsigma), 3)$

Figure 3.4: *Three transactions for Example 3.1.4. Note that, by Definition 3.7, the first witness of* T'_3 *is* $\mathsf{sig}_k^{**,1}(\mathsf{T}'_3)$*, while the second is* $\mathsf{sig}_k^{**,2}(\mathsf{T}'_3)$*.*

Example 3.1.3. With the transactions in Figure 3.3, we have $(\mathsf{T}_0, 1, t_0) \overset{v_0}{\rightsquigarrow} (\mathsf{T}_1, 1, t_1)$. Indeed, for item (a) we have that $\mathsf{T}_1.\mathsf{in}(1) = (\mathsf{T}_0\{\mathsf{wit} \mapsto \perp\}, 1)$; for item (b), $\mathsf{T}_1, 1 \models \lambda\varsigma.\mathsf{versig}(k,\varsigma)$; for item (c), $v_0 = \mathsf{val}(\mathsf{T}_0.\mathsf{out}(1))$. The other two items trivially hold, as there are no time constraints. We also have $(\mathsf{T}_0, 1, 2.1.2017) \overset{v_0}{\rightsquigarrow} (\mathsf{T}'_1, 1, 6.1.2017)$. To show that, we have to check also items (d) and (e). For item (d), we have that $6.1.2017 \geq \mathsf{T}'_1.\mathsf{absLock} = 5.1.2017$. For item (e), we have that $6.1.2017 - 2.1.2017 \geq \mathsf{T}'_1.\mathsf{relLock}(1) = 2$ days. □

Example 3.1.4. Consider the transactions in Figure 3.4. The signature in $\mathsf{T}'_3.\mathsf{wit}(1)$ is computed as follows:

$$
\begin{aligned}
\mathsf{sig}_k^{**,1}(\mathsf{T}'_3) &= (sig_k(**_1(\mathsf{T}'_3, **)), **) && \text{by Definition 3.6} \\
&= (sig_k(\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \perp\}, **), **) && \text{by Definition 3.5}
\end{aligned}
$$

We prove that, when verifying $(\mathsf{T}'_1, 1, t) \overset{1}{\rightsquigarrow} (\mathsf{T}'_3, 1, t')$, item (b) of Definition 3.11 holds, i.e. $\mathsf{T}'_3, 1 \models \mathsf{scr}(\mathsf{T}'_1.\mathsf{out}(1))$. To this purpose, let $\rho = \{\varsigma \mapsto (w, **)\}$, where $w = sig_k(\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, **)$. We have that:

$$\begin{aligned}
[\![\mathsf{versig}(k,\varsigma)]\!]_{\mathsf{T}'_3,1,\rho} &= \mathsf{ver}_k([\![\varsigma]\!]_{\mathsf{T}'_3,1,\rho}, \mathsf{T}'_3, 1) && \text{by Def. 3.9}\\
&= \mathsf{ver}_k((w,**), \mathsf{T}'_3, 1) && \rho(\varsigma) = (w,**)\\
&= ver_k(w, (**_1(\mathsf{T}'_3), **)) && \text{by Def. 3.6}\\
&= ver_k(w, (\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, **)) && \text{by Def. 3.5}\\
&= true && \text{by Def. of } w
\end{aligned}$$

We now show that w is *not* valid for the other witness, i.e. $(\mathsf{T}'_2, 1, t) \overset{2}{\not\rightsquigarrow} (\mathsf{T}''_3, 2, t')$, where $\mathsf{T}''_3 = \mathsf{T}'_3\{\mathsf{wit}(2) \mapsto \mathsf{sig}_k^{**,1}(\mathsf{T}'_3)\}$. Let $\rho = \{\varsigma \mapsto (w, **)\}$. Item (b) of Definition 3.11 does not hold:

$$\begin{aligned}
[\![\mathsf{versig}(k,\varsigma)]\!]_{\mathsf{T}''_3,2,\rho} &= \mathsf{ver}_k((w,**), \mathsf{T}''_3, 2) && \text{as above}\\
&= ver_k(w, (**_2(\mathsf{T}''_3), **)) && \text{by Def. 3.6}\\
&= ver_k(w, (\mathsf{T}''_3\{\mathsf{wit}(1) \mapsto 2\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, **)) && \text{by Def. 3.5}\\
&= false
\end{aligned}$$

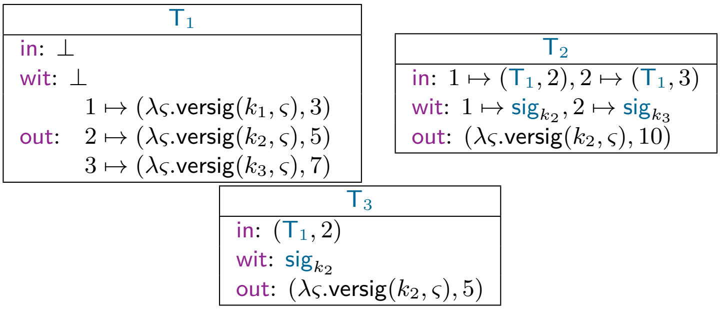
In the last equation, w is not a valid signature for $\mathsf{T}''_3\{\mathsf{wit}(1) \mapsto 2\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}$ because it is computed on $\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}$, and the two transactions differ on $\mathsf{wit}(1)$. □

3.1.6 Blockchain and consistency

In Definition 3.12 we model blockchains as sequences of *timed transactions* (T, t), where t represents the time when the transaction T has been added. Note that our definition is very permissive: for instance, it allows a blockchain to contain transactions which do not redeem any transactions, or double-spent transactions. We will rule out such *inconsistent* blockchains later on in Definition 3.15.

Definition 3.12 (Blockchain). *A blockchain* $\mathbf{B}$ *is a sequence* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$, *where* T_1 *is the only transaction with* $\mathsf{in} = \bot$, *and* $t_i \leq t_j$ *for all* $1 \leq i \leq j \leq n$.

We denote with $trans_{\mathbf{B}}$ *the set of transactions occurring in* $\mathbf{B}$, *and with* $time_{\mathbf{B}}(\mathsf{T}_i)$ *the time* t_i *of transaction* T_i *in* $\mathbf{B}$. *Given a transaction* T, *we define* $match_{\mathbf{B}}(\mathsf{T})$ *as the set of transactions* T_i *such that* $\mathsf{T}\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}_i\{\mathsf{wit} \mapsto \bot\}$.



T_1	
in:	$\bot$
wit:	$\bot$
out:	$1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_1,\varsigma), 3)$ $2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_2,\varsigma), 5)$ $3 \mapsto (\lambda\varsigma.\mathsf{versig}(k_3,\varsigma), 7)$

T_2	
in:	$1 \mapsto (\mathsf{T}_1, 2), 2 \mapsto (\mathsf{T}_1, 3)$
wit:	$1 \mapsto \mathsf{sig}_{k_2}, 2 \mapsto \mathsf{sig}_{k_3}$
out:	$(\lambda\varsigma.\mathsf{versig}(k_2,\varsigma), 10)$

T_3	
in:	$(\mathsf{T}_1, 2)$
wit:	sig_{k_2}
out:	$(\lambda\varsigma.\mathsf{versig}(k_2,\varsigma), 5)$

Figure 3.5: *Three transactions for Examples 3.1.5 to 3.1.7.*

Definition 3.13 (Unspent output)**.** *Let* $\mathsf{B} = (\mathsf{T}_1, t_1)\cdots(\mathsf{T}_n, t_n)$ *be a blockchain. We say that the output* j *of transaction* T_i *is* unspent *in* B *whenever:*

$$\forall i' \leq n, j' \in \mathbb{N} \quad : \quad (\mathsf{T}_i, j, t_i) \not\rightsquigarrow (\mathsf{T}_{i'}, j', t_{i'})$$

Given a blockchain B*, we define:*

- $UTXO_{\mathsf{B}}$*, the* Unspent Transaction Output *of* B*, as the set of pairs* (T_i, j) *such that output* j *of* T_i *is unspent in* B*.*
- $\mathsf{val}(\mathsf{B})$*, the* value *of* B*, as the sum of the values of all outputs in its UTXO.*

Example 3.1.5. Consider the transactions in Figure 3.5, and let $\mathsf{B} = (\mathsf{T}_1, 0)(\mathsf{T}_2, t_2)$. We have that $(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2)$ and $(\mathsf{T}_1, 3, 0) \overset{7}{\rightsquigarrow} (\mathsf{T}_2, 2, t_2)$, while the other outputs are unspent. Hence, the UTXO of B is $\{(\mathsf{T}_1, 1), (\mathsf{T}_2, 1)\}$. □

The following definition establishes when (T, t) is a *consistent update* of B.

Definition 3.14 (Consistent update)**.** *We write* $\mathsf{B} \triangleright (\mathsf{T}, t)$ *iff either* $\mathsf{B} = \varepsilon$*,* T *is initial and* $t = 0$*, or, for all* $i \in \operatorname{dom}(\mathsf{T}.\mathsf{in})$*:*

$$\begin{array}{ll}
\{\mathsf{T}'_i\} = match_{\mathsf{B}}(fst(\mathsf{T}.\mathsf{in}(i))) & \textit{(redeemed transaction)} \\
o_i = snd(\mathsf{T}.\mathsf{in}(i)) & \textit{(redeemed output index)} \\
t'_i = time_{\mathsf{B}}(\mathsf{T}'_i) & \textit{(time when } \mathsf{T}'_i \textit{ was added to } \mathsf{B}\textit{)} \\
v_i = \mathsf{val}(\mathsf{T}'_i.\mathsf{out}(o_i)) & \textit{(value of the redeemed output)}
\end{array}$$

the following conditions hold:

(1) $\forall i \in \operatorname{dom} \mathsf{T}.\mathsf{in} : (\mathsf{T}'_i, o_i) \in UTXO_{\mathbf{B}}$

(2) $\forall i \in \operatorname{dom} \mathsf{T}.\mathsf{in} : (\mathsf{T}'_i, o_i, t'_i) \overset{v_i}{\rightsquigarrow} (\mathsf{T}, i, t)$

(3) $\sum \{v_i \mid i \in \operatorname{dom} \mathsf{T}.\mathsf{in}\} \;\geq\; \sum \{\mathsf{val}(\mathsf{T}.\mathsf{out}(j)) \mid j \in \operatorname{dom} \mathsf{T}.\mathsf{out}\}$

(4) $\mathbf{B} = \mathbf{B}'(\mathsf{T}', t') \implies t \geq t'$

Firstly, for each $\mathsf{T}.\mathsf{in}(i)$ we obtain the singleton $\{\mathsf{T}'_i\}$ from the blockchain, using $match_{\mathbf{B}}$, such that $fst(\mathsf{T}.\mathsf{in}(i))\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}'_i\{\mathsf{wit} \mapsto \bot\}$. The update is inconsistent if $match_{\mathbf{B}}(fst(\mathsf{T}.\mathsf{in}(i)))$ is not a singleton for some i. Condition (1) requires that the redeemed outputs are currently unspent in $\mathbf{B}$. Condition (2) asks that each input of T redeems an output of a transaction in $\mathbf{B}$. Condition (3) requires that the sum of the values of the outputs of T is not greater than the total value it redeems. Finally, (4) requires that the time of T is greater than or equal to the time of the last transaction in $\mathbf{B}$.

Example 3.1.6. Consider again the transactions in Figure 3.5, and let $\mathbf{B} = (\mathsf{T}_1, 0)$. We prove that $\mathbf{B} \rhd (\mathsf{T}_2, t_2)$. Let $o_1 = 2$, $o_2 = 3$, $t'_1 = t_2 = 0$, $v_1 = 5$, $v_2 = 7$. We now prove that the conditions of Definition 3.14 are satisfied. For condition (1), note that both $(\mathsf{T}_1, 2)$ and $(\mathsf{T}_1, 3)$ are unspent, according to Definition 3.13. For condition (2), note that:

$$(\mathsf{T}_1, 2, 0) \overset{v_1}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2) \qquad (\mathsf{T}_1, 3, 0) \overset{v_2}{\rightsquigarrow} (\mathsf{T}_2, 2, t_2)$$

hold, according to Definition 3.11. Finally, for condition (3), we have that:

$$\sum \{v_i \mid i \in \{1, 2\}\} = 5 + 7 \;\geq\; \sum \{\mathsf{val}(\mathsf{T}_2.\mathsf{out}(j)) \mid j \in \operatorname{dom} \mathsf{T}_2.\mathsf{out}\} = 10$$

Therefore, (T_2, t_2) is a consistent update of $\mathbf{B}$. □

Example 3.1.7 (Double spending)**.** Consider again the transactions in Figure 3.5, and let $\mathbf{B} = (\mathsf{T}_1, 0)(\mathsf{T}_2, t_2)$. We prove that (T_3, t_3) is *not* a consistent update of $\mathbf{B}$. Although condition (2) of Definition 3.14 holds:

$$(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_3, 1, t_3)$$

we have that condition (1) is *not* satisfied. In fact, according to Definition 3.13, $(\mathsf{T}_1, 2)$ is already spent in $\mathbf{B}$ because

$$(\mathsf{T}_1, 2, 0) \overset{5}{\rightsquigarrow} (\mathsf{T}_2, 1, t_2)$$

holds and both T_1 and T_2 are in $\mathbf{B}$. Since T_3 is trying to spend an output already spent, this transaction should not be appended to $\mathbf{B}$. $\square$

We now define when a blockchain is consistent. Intuitively, consistency holds when the blockchain has been constructed, starting from the empty one, by appending consistent updates, only. The actual definition is given by induction.

Definition 3.15 (Consistency). *We say that a blockchain* $\mathbf{B}$ *is consistent if either* $\mathbf{B} = \varepsilon$*, or* $\mathbf{B} = \mathbf{B}'(\mathsf{T}, t)$ *with* $\mathbf{B}'$ *consistent and* $\mathbf{B}' \rhd (\mathsf{T}, t)$*.*

Note that the empty blockchain is consistent; the blockchain with a single transaction (T_1, t_1) is consistent iff T_1 is initial and $t_1 = 0$. The transaction T_1 models the first transaction in the *genesis block* (as discussed in Section 7.3, we are abstracting away the *coinbase* transactions, which forge new bitcoins).

We now establish some basic properties of consistent blockchains. Theorem 3.1.1 states that, in a consistent blockchain, the inputs of a transaction point backwards to the output of some transaction in the blockchain.

Lemma 3.1.1. If $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ is consistent, then:

$$\begin{gathered} \forall i \in 2 \dots n : \forall (\mathsf{T}, h) \in \operatorname{ran}(\mathsf{T}_i.\mathsf{in}) : \exists j < i : \\ \mathsf{T}_j\{\mathsf{wit} \mapsto \perp\} = \mathsf{T} \ \wedge \ h \in \operatorname{dom}(\mathsf{T}_j.\mathsf{out}) \end{gathered}$$

Proof. By Definition 3.15, (T_i, t_i) is a consistent update of $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_{i-1}, t_{i-1})$. The thesis follows from condition (2) of Definition 3.14. $\square$

The following Theorem establishes that a transaction output cannot be redeemed twice in a consistent blockchain.

Theorem 3.1.2 (No double spending). If $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ is consistent, then:

$$\forall i \neq j \in 1 \dots n \quad : \quad \operatorname{ran}(\mathsf{T}_i.\mathsf{in}) \cap \operatorname{ran}(\mathsf{T}_j.\mathsf{in}) = \emptyset$$

Proof. Let $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ be consistent. By contradiction, assume that there exist $i < j$ and i', j' such that $\mathsf{T}_i.\mathsf{in}(i') = \mathsf{T}_j.\mathsf{in}(j')$. By consistency, there exist h, h' such that $(\mathsf{T}_h\{\mathsf{wit} \mapsto \perp\}, h') = \mathsf{T}_i.\mathsf{in}(i')$. Since $\mathbf{B}_{1..i-1} \rhd (\mathsf{T}_i, t_i)$, then by item (2) of Definition 3.14 it must be $(\mathsf{T}_h, h', t_h) \rightsquigarrow (\mathsf{T}_i, i', t_i)$. Hence, by Definition 3.13 it follows that (T_h, h') is already *spent* in $\mathbf{B}$. Since $\mathbf{B}_{1..j-1} \rhd (\mathsf{T}_j, t_j)$, by item (1) of Definition 3.14, (T_h, h') must be *unspent* — contradiction. $\square$

The following theorem states that there can be at most a single match of an arbitrary transaction within a consistent blockchain. This implies that the in field of an arbitrary transaction points at most to one transaction output within the blockchain.

Lemma 3.1.3. If B is consistent, then for all transactions T, $match_{\mathsf{B}}(\mathsf{T})$ contains at most one element.

Proof. Let $\mathsf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ be consistent. By contradiction, assume that $\mathsf{T}_i, \mathsf{T}_j \in match_{\mathsf{B}}(\mathsf{T})$, with $\mathsf{T}_i \neq \mathsf{T}_j$ (and so, $i \neq j$). By Definition 3.12 it must be $\mathsf{T}_i\{\mathsf{wit} \mapsto \perp\} = \mathsf{T}\{\mathsf{wit} \mapsto \perp\} = \mathsf{T}_j\{\mathsf{wit} \mapsto \perp\}$, hence in particular $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in}$. There are two cases. If $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in} = \perp$, then by Definition 3.12 B is not a blockchain, since $i \neq j$. Hence, $\operatorname{ran}(\mathsf{T}_i.\mathsf{in}) \cap \operatorname{ran}(\mathsf{T}_j.\mathsf{in}) = \operatorname{ran}(\mathsf{T}_i.\mathsf{in}) \neq \emptyset$. By Theorem 3.1.2, this cannot happen because B is consistent — contradiction. □

Theorem 3.1.4 ensures that all the transactions on a consistent blockchain are pairwise distinct, even when neglecting their witnesses.

Lemma 3.1.4. If $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ is consistent, then:

$$\forall i \neq j \in 1 \ldots n \quad : \quad \mathsf{T}_i\{\mathsf{wit} \mapsto \perp\} \neq \mathsf{T}_j\{\mathsf{wit} \mapsto \perp\}$$

Proof. Straightforward from Theorem 3.1.3, taking $\mathsf{T} = \mathsf{T}_j$. □

The following theorem states that the overall value of a blockchain decreases as the blockchain grows. This is because our model does not keep track of the *coinbase transactions*, which in Bitcoin allow miners to collect transaction fees (the difference between inputs and outputs of a transaction), and block rewards.

Theorem 3.1.5 (Non-increasing value)**.** Let B be a consistent blockchain, and let B' be a non-empty prefix of B. Then, $\mathsf{val}(\mathsf{B}') \geq \mathsf{val}(\mathsf{B})$.

Proof. Let $\mathsf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$. By contradiction, there exists some $i < n$ such that, given $\mathsf{B}_i = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_i, t_i)$:

$$\mathsf{val}(\mathsf{B}_i) \ < \ \mathsf{val}(\mathsf{B}_i(\mathsf{T}_{i+1}, t_{i+1}))$$

Let U_i and U_{i+1} be the UTXOs of B_i and of $\mathsf{B}_i(\mathsf{T}_{i+1}, t_{i+1})$, respectively, and let $U = U_i \cap U_{i+1}$. Since $\mathsf{val}(U_i) < \mathsf{val}(U_{i+1})$, then it must be $\mathsf{val}(U_i \setminus U) < \mathsf{val}(U_{i+1} \setminus U)$. The set $U_i \setminus U$ contains the outputs redeemed by T_{i+1}, while the set $U_{i+1} \setminus U$ contains exactly the outputs in

T_{AB}
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\perp$
out: $(\lambda \varsigma_{\mathsf{A}} \varsigma_{\mathsf{B}}.\mathsf{versig}(k_{\mathsf{A}} k_{\mathsf{B}}, \varsigma_{\mathsf{A}} \varsigma_{\mathsf{B}}), 1.1\text{B})$

T_{BC}
in: $(\mathsf{T}_{\mathsf{AB}}, 1)$
wit: $\perp$
out: $1 \mapsto (\lambda \varsigma_{\mathsf{B}}.\mathsf{versig}(k_{\mathsf{B}}, \varsigma_{\mathsf{B}}), 0.1\text{B})$ $2 \mapsto (\lambda \varsigma_{\mathsf{C}}.\mathsf{versig}(k_{\mathsf{C}}, \varsigma_{\mathsf{C}}), 1\text{B})$

Figure 3.6: *Transactions of the chain contract.*

T_{i+1}. Since $\mathbf{B}$ is consistent, then $\mathbf{B}_i \rhd (\mathsf{T}_{i+1}, t_{i+1})$. Then, by Definition 3.14, for each $k \in \operatorname{dom} \mathsf{T}_{i+1}.\mathsf{in}$, there exists a unique $j \leq i$ such that, given $o_k = snd(\mathsf{T}_{i+1}.\mathsf{in}(k))$ and $v_k = \mathsf{val}(\mathsf{T}_j.\mathsf{out}(o_k))$:

$$(\mathsf{T}_j, o_k, t_j) \overset{v_k}{\rightsquigarrow} (\mathsf{T}_{i+1}, k, t_{i+1})$$

Then, by item (3) of Definition 3.14:

$$\begin{aligned} \mathsf{val}(U_i \setminus U) &= \sum \{v_k \mid k \in \operatorname{dom} \mathsf{T}_{i+1}.\mathsf{in}\} \\ &\geq \sum \{\mathsf{val}(\mathsf{T}_{i+1}.\mathsf{out}(h)) \mid h \in \operatorname{dom} \mathsf{T}_{i+1}.\mathsf{out}\} = \mathsf{val}(U_{i+1} \setminus U) \end{aligned}$$

while we assumed $\mathsf{val}(U_i \setminus U) < \mathsf{val}(U_{i+1} \setminus U)$ — contradiction. □

Note that the scripting language and its semantics are immaterial in all the statements above. Actually, proving these results never involves checking condition (b) of Definition 3.11. Of course, the choice of the scripting language affects the expressiveness of the smart contracts built upon Bitcoin.

3.2 Example: static chains of transactions

We now formally specify in our model a simple smart contract, which illustrates the impact of Segregated Witness on the expressiveness of Bitcoin contracts [112].

A participant A wants to send an indirect payment of 1B to C, routing it through B. To authorize the payment, B wants to keep a fee of 0.1B. However, A is afraid that B will keep all the money for himself, so she exploits the following contract. She creates a *chain* of transactions, as shown in Figure 3.6. The transaction T_{AB} transfers 1.1B from A to B (but it is not signed by A, yet), while T_{BC} transfers 1B from B to C. We assume that $(\mathsf{T}_{\mathsf{A}}, 1)$ is a transaction output redeemable by A through her key k_{A}, and that k_{B} is the key of B.

The protocol of A is the following: A starts by asking B for his signature on T_{BC}, ensuring that C will be paid. After receiving and verifying the signature, A puts T_{AB} on the blockchain, adding her signature on the wit field. Then, she also appends T_{BC}, replacing the wit field with her signature and B's one. Since A takes care of publishing the transactions, the behaviour of B consists just in sending his signature on T_{BC}.

Remarkably, this contract relies on the SegWit feature: indeed, without SegWit it no longer works. We can disable SegWit by changing our model as follows:

- in Definition 3.3, we no longer require that $\forall i \in \operatorname{dom} \mathsf{in} : \mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \perp$
- in Definition 3.11, we replace item (a) with the condition: $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{T}, i)$
- in Definition 3.12, we let $\mathit{match}_{\mathbf{B}}(\mathsf{T}) = \{\mathsf{T}\}$ if T occurs in $\mathbf{B}$, empty otherwise.

To see why disabling SegWit breaks the contract, assume that the transaction $\mathsf{T} = \mathsf{T}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{AB}})\}$ is unspent on the blockchain, when participant A attempts to append also $\mathsf{T}' = \mathsf{T}_{\mathsf{BC}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{BC}})\, \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}_{\mathsf{BC}})\}$. To be a consistent update, by item (2) of Definition 3.14 we must have (for some $t_1 \leq t_2$):

$$(\mathsf{T}, 1, t_1) \overset{1 \text{₿}}{\rightsquigarrow} (\mathsf{T}', 1, t_2) \tag{3.1}$$

For this, all the conditions in Definition 3.11 must hold. However, since we have disabled SegWit, for item (a) we no longer check that:

$$\mathsf{T}'.\mathsf{in}(1) \;=\; (\mathsf{T}\{\mathsf{wit} \mapsto \perp\}, 1)$$

but instead we need to check the condition:

$$\tilde{\mathsf{T}}'.\mathsf{in}(1) \;=\; (\tilde{\mathsf{T}}, 1) \tag{3.2}$$

where the transactions $\tilde{\mathsf{T}}, \tilde{\mathsf{T}}'$ correspond to the non-SegWit versions of T, T', i.e. their in fields point to their actual parents, according to the new Definition 3.3.

Hence, condition (3.2) checks the equality between $\tilde{\mathsf{T}}_{\mathsf{AB}}$ (the transaction in the input of $\tilde{\mathsf{T}}'$) and $\tilde{\mathsf{T}}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\tilde{\mathsf{T}}_{\mathsf{AB}})\}$ (the transaction $\tilde{\mathsf{T}}$). Note that all the fields of the second transaction — but the wit field — are equal to those of the first transaction. Instead, the witness of $\tilde{\mathsf{T}}_{\mathsf{AB}}$ is $\perp$, while the one of $\tilde{\mathsf{T}}$ contains the signature of A. This difference in the wit field is ignored with the SegWit semantics, while it is discriminating for the older version of Bitcoin.

A naïve attempt to amend the contract would be to set the input field of $\tilde{\mathsf{T}}'$ to $\tilde{\mathsf{T}}$. However, this would invalidate the signature of A on $\tilde{\mathsf{T}}'$.

3.3 Compiling to standard Bitcoin transactions

We now sketch how to compile the transactions of our abstract model into concrete Bitcoin transactions. In particular, we aim at producing *standard* Bitcoin transactions, which respect further constraints on their fields [85]. This is crucial, because non-standard transactions are mostly discarded by the Bitcoin network.

The compiler, and a in-depth documentation are available at [81]. It produces output scripts of the following kinds, which are all allowed in standard transactions:

Pay to Public Key Hash (P2PKH) takes as parameters a public key and a signature, and checks that (i) the hash of the public key matches the hash hardcoded in the script; (ii) the signature is verified against the public key.

Pay to Script Hash (P2SH) contains only a hash (say, h). The actual script $\lambda \boldsymbol{x}.e$ — which is *not* required to be standard — is contained instead in the wit field of the redeeming transaction, alongside with the actual parameters $\boldsymbol{k}$. The evaluation succeeds if $H(\lambda \boldsymbol{x}.e) = h$ and $(\lambda \boldsymbol{x}.e)\boldsymbol{k}$ evaluates to *true*. The only constraint imposed by P2SH is on the size of the script, which is limited to the size of a stack element (520 bytes).

OP_RETURN allows to put up to 80 bytes of data in an output script, making the output unredeemable.

We compile the scripts of the form $\lambda\varsigma.\mathsf{versig}(k, \varsigma)$ to P2PKH, and those of the form $\lambda.k$ to OP_RETURN. All other scripts are compiled to P2SH when they comply with the size constraint, otherwise compilation fails. In this way, our compiler always produces standard transactions.

Our compiler exploits the *alternative stack* as temporary storage of the variable values. In this way we cope with the stack-based nature of the Bitcoin scripting language. For instance, for the script $\lambda x.\mathsf{H}(x) = \mathsf{H}(x+1)$, the variable x is pushed on the alternative stack beforehand, then duplicated and copied in the main stack before each operation involving x.

There exist other standard scripts: P2PK and MULTISIG. P2PK is considered obsolete and replaced by P2PKH, while MULTISIG has some limitations (e.g. in the number of keys) that are overcome using P2SH to express the same semantics.

Related works Several works have proposed to use Bitcoin beyond the sole purpose of exchanging currency, by exploiting the flexibility of its scripting language. They propose to implement smart contracts, intended as sets of protocols of the participants involved in them.

Smart contracts requiring external state, namely oracles [87] and escrows [84], can be easily implemented using multi-signature transactions. Such implementations, however, rely on trusted third parties. The work [4] showed that Bitcoin can be used to implement timed commitments through deposit transactions. The commitments are then used to perform multiparty computations [1], such as calculating the winner of a lottery. The main drawback of this approach is indeed the deposit, which grows quadratically with the number of participants. More recently, [62] and [23] have proposed lottery smart contracts that require, respectively, zero and constant (≥ 0) deposit. However, [62] requires the computation of an exponential number of signature w.r.t. the number of participants, while [23] only a quadratic one. The work [12] proposed a contingent payment protocol that can be implemented relying only on standard Bitcoin transaction. It allows to sell solutions for a class of NP problems (e.g. the factorization of a number), the use of zero-knowledge proofs ensure its correctness to the buyer.

Chapter 4

Extending Bitcoin with Neighborhood Covenants

Bitcoin is a decentralised infrastructure to transfer cryptocurrency between users. The log of all the currency transactions is recorded in a public, append-only, distributed data structure, called blockchain. Bitcoin implements a model of computation called *Unspent Transaction Output (UTXO)*: each transaction holds an amount of currency, and specifies conditions under which this amount can be redeemed by a subsequent transaction, which spends the old one. Compared to the *account-based* model, implemented e.g. by Ethereum, the UTXO model does not require a shared mutable state: the current state is given just by the set of unspent transaction outputs on the blockchain. While, on the one hand, this design choice fits well with the inherent concurrency of transactions, on the other hand the lack of a shared mutable state substantially complicates leveraging Bitcoin to implement *contracts*, i.e. protocols which transfer cryptocurrency according to programmable rules.

The literature has shown that Bitcoin contracts support a surprising variety of use cases, including e.g. crowdfunding [86, 9], lotteries and other gambling games [1, 23, 9, 26, 57, 62], contingent payments [12], micropayment channels [122, 9], and other kinds of fair computations [4, 56]. Despite this apparent richness, the fact is that Bitcoin contracts cannot express most of the use cases that are mainstream in other blockchain platforms (e.g., decentralised finance). There are several factors that limit the expressiveness of Bitcoin contracts. Among them, the crucial one is the script language used to express the redeeming conditions within transactions. This language only features a limited set of logic, arithmetic, and cryptographic operators, but it does not have loops, and it cannot access parts of the spent and of the redeeming transaction.

Several extensions of the Bitcoin script language have been proposed, with the aim to improve the expressiveness of Bitcoin contracts, while adhering to the UTXO model. Among these extensions, *covenants* are a class of script operators that allow a transaction to constrain how its funds can be used by the redeeming transactions. Covenants may also be recursive, by requiring the script of the redeeming transaction to contain the same covenant of the spent one. As noted by [66], recursive covenants would allow to implement Bitcoin contracts that execute state machines, by appending transactions to trigger state transitions.

Although the first proposals of covenants date back at least to 2013 [114], and that they are supported by Bitcoin fork "Bitcoin Cash" [110], their inclusion into Bitcoin is still uncertain, mainly because of the extremely cautious approach to implement changes to Bitcoin [102]. Still, the emerging of Bitcoin layer-2 protocols, like e.g. the Lightning Network [122], has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [123, 70]), and by the incorporation of covenants in Liquid's extensions to Bitcoin Script [117].

We propose a variant of covenants, named *neighbourhood covenants*, which can inspect not only the redeeming transaction, but also the siblings and the parent of the spent one. This extension preserves the basic UTXO design of Bitcoin, adding only a few opcodes to its script language, which is kept efficient, loop-free, and *non* Turing-complete. Still, neighbourhood covenants significantly increase the expressiveness of Bitcoin as a smart contracts platform, allowing to execute arbitrary smart contracts by appending a *chain* of transactions to the blockchain. Technically, we prove that neighbourhood covenants make Bitcoin Turing-complete.

Although this expressiveness result is of theoretical interest, in itself it does not enable an efficient implementation of tokens. To recover efficiency, we implement our new use cases using scripts which exploit covenants.

4.1 Neighbourhood covenants

To extend Bitcoin with neighbourhood covenants, we amend the model of pure Bitcoin in the previous Chapter as follows:

- in transactions, we add a field to outputs, making them records of the form $\{\mathsf{arg} : \boldsymbol{a}, \mathsf{scr} : e, \mathsf{val} : v\}$, where $\boldsymbol{a}$ is a sequence of values;

- in scripts, we add operators to access all the outputs of the redeeming transaction, and a relevant subset of those of the sibling and parent transactions (by contrast, pure Bitcoin scripts can only access the redeeming transaction, and only as a whole, to verify it against a signature);
- in scripts, we add operators for covenants.

We now formalize our Bitcoin extension. We use o to refer to the following transaction outputs:

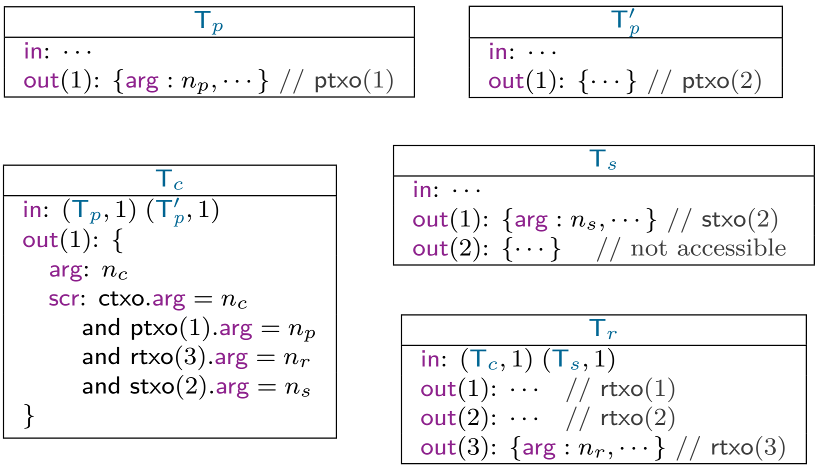
$$\begin{array}{rlll} \mathsf{o} & ::= & \mathsf{rtxo}(e) & \text{output of the redeeming tx} \\ & \mid & \mathsf{stxo}(e) & \text{output of a sibling tx} \\ & \mid & \mathsf{ptxo}(e) & \text{output of a parent tx} \end{array}$$

More precisely, consider the case where a transaction output (T', j) is redeemed by a transaction T, through its i-th input. When used within the script of (T', j): $\mathsf{rtxo}(n)$ refers to the n-th output of T; $\mathsf{stxo}(n)$ refers to the output redeemed by the n-th input of T; $\mathsf{ptxo}(n)$ refers to the output redeemed by the n-th input of T'. In Figure 4.1, we exemplify these outputs in relation to the transaction T_c. The semantics of o is defined in the first line of Figure 4.2; its result is a pair (T, n).

We extend scripts as follows, where $\mathsf{f} \in \{\mathsf{arg}, \mathsf{val}\}$:

$$\begin{array}{rlll} e & ::= & \cdots \mid \mathsf{o.f} & \text{field of a tx output} \\ & & \mid \mathsf{verscr}(e, \mathsf{o}) & \text{basic covenant} \\ & & \mid \mathsf{verrec}(\mathsf{o}) & \text{recursive covenant} \\ & & \mid \mathsf{inidx} & \text{index of redeeming tx input} \\ & & \mid \mathsf{outidx} & \text{index of redeemed tx output} \\ & & \mid \mathsf{inlen}(\mathsf{o}) & \text{number of inputs} \\ & & \mid \mathsf{outlen}(\mathsf{o}) & \text{number of outputs} \\ & & \mid \mathsf{txid}(\mathsf{o}) & \text{hash of (tx,output)} \end{array}$$

The script $\mathsf{o.f}$ gives access to the field f of a transaction output o (where f is either arg or val). The basic covenant $\mathsf{verscr}(e, \mathsf{o})$ checks that the script in the transaction output o is syntactically equal to e (note that e is not evaluated). The "recursive" covenant $\mathsf{verrec}(\mathsf{o})$ checks that the script in o is syntactically equal to the script which is currently being evaluated. We call it recursive because it is a covenant that enforce another one equal to itself. The operators inidx and outidx evaluate, respectively, to the index of the redeeming input and redeemed output. We call *current transaction output* (ctxo) the transaction output which includes the script which is



```
T_p
in: ···
out(1): {arg : n_p, ···} // ptxo(1)

T'_p
in: ···
out(1): {···} // ptxo(2)

T_c
in: (T_p, 1) (T'_p, 1)
out(1): {
   arg: n_c
   scr: ctxo.arg = n_c
      and ptxo(1).arg = n_p
      and rtxo(3).arg = n_r
      and stxo(2).arg = n_s
}

T_s
in: ···
out(1): {arg : n_s, ···} // stxo(2)
out(2): {···}   // not accessible

T_r
in: (T_c, 1) (T_s, 1)
out(1): ···   // rtxo(1)
out(2): ···   // rtxo(2)
out(3): {arg : n_r, ···} // rtxo(3)
```

Figure 4.1: *Accessing transaction outputs through a script.*

currently being evaluated, i.e.:

$$\mathsf{ctxo} \triangleq \mathsf{stxo}(\mathsf{inidx})$$

The scripts $\mathsf{inlen}(\mathsf{o})$ and $\mathsf{outlen}(\mathsf{o})$ evaluate, respectively, to the number of inputs and to the number of outputs of the transaction containing o. Finally, $\mathsf{txid}(\mathsf{o})$ evaluates to a unique identifier of the transaction output o.

Example 4.1.1. Consider the transactions in Figure 4.1. The script in $\mathsf{T}_c.\mathsf{out}(1)$ checks that (i) the arg field of ctxo, i.e. the same output which contains the script, equals to n_c; (ii) the arg field of $\mathsf{ptxo}(1)$, i.e. the parent transaction output redeemed by $\mathsf{T}_c.\mathsf{in}(1)$, equals to n_p; (iii) the arg field of $\mathsf{rtxo}(3)$, i.e. the third output of the redeeming transaction T_r, equals to n_r; (iv) the arg field of $\mathsf{stxo}(2)$, i.e. the sibling transaction output redeemed by $\mathsf{T}_r.\mathsf{in}(2)$, equals to n_s. Note that, in general, any script used in T_c can not access the parents of T_p and those of T_s — and in general all the transactions which are farther than those shown in the figure. $\diamond$

Figure 4.2 defines the semantics of extended scripts. As in Chapter 3, the function $[\![\cdot]\!]$ takes as parameters the redeeming transaction T and the index i of the redeeming input. We denote with $\equiv$ syntactic equality between two scripts, i.e. $e \equiv e'$ is 1 when e and e' are exactly the same, 0 otherwise.

$$[\![\mathsf{rtxo}(e)]\!]_{\mathsf{T},i} = (\mathsf{T}, [\![e]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{stxo}(e)]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{in}([\![e]\!]_{\mathsf{T},i})$$

$$[\![\mathsf{ptxo}(e)]\!]_{\mathsf{T},i} = \mathsf{T}'.\mathsf{in}([\![e]\!]_{\mathsf{T},i}) \text{ if } \mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j) \qquad [\![\mathsf{o}.\mathsf{f}]\!]_{\mathsf{T},i} = [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{f}$$

$$[\![\mathsf{verscr}(e, \mathsf{o})]\!]_{\mathsf{T},i} = e \equiv [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{scr} \qquad [\![\mathsf{verrec}(\mathsf{o})]\!]_{\mathsf{T},i} = \mathsf{T}.\mathsf{in}(i).\mathsf{scr} \equiv [\![\mathsf{o}]\!]_{\mathsf{T},i}.\mathsf{scr}$$

$$[\![\mathsf{txid}(\mathsf{o})]\!]_{\mathsf{T},i} = H([\![\mathsf{o}]\!]_{\mathsf{T},i}) \qquad [\![\mathsf{outidx}]\!]_{\mathsf{T},i} = j \text{ if } \mathsf{T}.\mathsf{in}(i) = (\mathsf{T}, j) \qquad [\![\mathsf{inidx}]\!]_{\mathsf{T},i} = i$$

$$[\![\mathsf{outlen}(\mathsf{o})]\!]_{\mathsf{T},i} = |\mathsf{T}'.\mathsf{out}| \qquad [\![\mathsf{inlen}(\mathsf{o})]\!]_{\mathsf{T},i} = |\mathsf{T}'.\mathsf{in}| \text{ if } [\![\mathsf{o}]\!]_{\mathsf{T},i} = (\mathsf{T}', j)$$

Figure 4.2: *Semantics of neighbourhood covenants (extending Figure 3.2).*

Turing completeness Our neighbourhood covenants make Bitcoin Turing-complete. To prove this, we describe how to simulate in the extended Bitcoin any counter machine [41], a well-known Turing-complete computational model. A *counter machine* is a pair (n, s), where $n \in \mathbb{N}$ is the number of integer registers of the machine, and s is a sequence of instructions. Instructions are the following: inc i increments register i, dec i decrements it, zero i sets it to zero, if $i \neq 0$ goto j conditionally jumps to instruction j when register i is not zero, halt terminates the machine. The state of a counter machine (n, s) is a tuple $(v_1, \ldots, v_n, p)$ where each v_i represents the current value of register i, and p is the number of the next instruction to execute (i.e., the program counter). To exploit the currency transfer capabilities of Bitcoin, we slightly extend the counter machine model, by requiring that the machine has an initial ₿ balance which, upon termination, is transferred to the user A if the content of the first register is 0, or to B otherwise. This allows the machine to execute programs that transfer bitcoins. We call this extended model *UTXO-counter machine.*

Theorem 4.1.1. Neighbourhood covenants can simulate any UTXO-counter machine. Hence, they are Turing-complete.

Proof. (*sketch*) We represent the state of the counter machine as a single transaction having one output. We simulate an execution step by appending a new transaction, which redeems the output representing the old state, and transfers its balance to a new output representing the new state. This is done until the machine halts, at which point we transfer its balance to the user determined by the final registers state. We remark that this simulation is made possible by the use of unbounded integers in our model, while Bitcoin only supports 32-bit integers.

We represent the machine state in the arg field of the transaction output o, storing it as a sequence of integers. As a shorthand, we write

$\mathsf{o.r}_i$ for $\mathsf{o.arg}.i$, and $\mathsf{o.p}$ for $\mathsf{o.arg}.(n+1)$. To simulate the execution steps of the machine, we define the script e_{CM}, which checks that the new transaction T indeed represents the next state. More in detail, we first check whether the instruction in s at position $\mathsf{ctxo.p}$ is halt, in which case we require that T distributes the balance to users in the intended manner, depending on the current state. When $\mathsf{ctxo.p}$ points to any other instruction, we start by requiring that T has only one output, the same balance, and the same script. We use a recursive covenant verrec on the redeeming transaction to ensure the last part. Then, we check that the new state in $\mathsf{T.out}(1).\mathsf{arg}$ agrees with the counter machine semantics, by cases on the instruction pointed to by $\mathsf{o.p}$. If the instruction is $\mathsf{inc}\ i$, then we require $\mathsf{rtxo}(1).\mathsf{r}_i = \mathsf{ctxo.r}_i + 1$, $\mathsf{rtxo}(1).\mathsf{r}_k = \mathsf{ctxo.r}_k$ for all $k \neq i$, and $\mathsf{rtxo}(1).\mathsf{p} = \mathsf{ctxo.p} + 1$. The $\mathsf{dec}\ i$ and $\mathsf{zero}\ i$ cases are analogous. For the instruction $\mathsf{if}\ i \neq 0\ \mathsf{goto}\ j$, we check the value of $\mathsf{ctxo.r}_i$: if nonzero, we require that $\mathsf{rtxo}(1).\mathsf{p} = j$, otherwise that $\mathsf{rtxo}(1).\mathsf{p} = \mathsf{ctxo.p} + 1$. In both cases, we require that $\mathsf{rtxo}(1).\mathsf{r}_k = \mathsf{ctxo.r}_k$ for all k.

Users start the simulation by appending to the blockchain a transaction T_0 having one output with the desired value, the script e_{CM}, and a sequence of $n+1$ zeros as arg. After that, the balance is effectively locked inside the transaction, and the only way to transfer it back to the users is to simulate all the steps of the machine, until it halts. So, our simulated execution is a form of *secure multiparty computation* [5, 75, 46]. □

Although, for simplicity, we use UTXO-counter machines just to transfer funds to A or B upon termination, it would be easy to generalise the computational model and simulation technique to encompass *interactive* computations, which at run-time can receive inputs and perform currency transfers. Doing so, we can execute arbitrary smart contracts, with the same expressiveness of Turing-complete smart contracts platforms. In principle, we could craft a smart contract which implements user-defined tokens in an account-based fashion: this contract would record the balance of the tokens of all users, and execute token actions. However, in practice this construction would be highly inefficient: performing a single token transfer would require to append a large number of transactions, and to pay the related fees.

4.2 Use cases

We illustrate the expressive power of our extension through a series of use cases, which, at the best of our knowledge, cannot be expressed in Bitcoin. We denote with $\mathsf{U}_{\mathsf{A}}^{v฿}$ an unspent transaction output $\{\mathsf{arg} : \varepsilon, \mathsf{scr} :$

T_i
in: $\mathsf{U}^{v_i\text{₿}}_{\mathsf{A}_i}$
wit: $\cdots$
out: $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : \mathit{CF},\ \mathsf{val} : v_i\text{₿}\}$

T_Z
in: $(\mathsf{T}_1, 1) \cdots (\mathsf{T}_n, 1)$
wit: $\mathit{sig}_\mathsf{Z}(\mathsf{T}_\mathsf{Z}) \cdots \mathit{sig}_\mathsf{Z}(\mathsf{T}_\mathsf{Z})$
out: $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : \mathsf{versig}(\mathsf{Z}, \mathsf{rtxo}(1).\mathsf{wit}),\ \mathsf{val} : v'\text{₿}\}$

Figure 4.3: *Transactions for the crowdfunding contract.*

$\mathsf{versig}(\mathsf{A}, \mathsf{rtx}.\mathsf{wit}),\ \mathsf{val} : v\text{₿}\}$, where ε denotes the empty sequence (we will usually omit arg when empty).

4.2.1 Crowdfunding

Assume that a start-up Z wants to raise funds through a crowdfunding campaign. The target of the campaign is to gather at least v₿ by time t. The contributors want the guarantee that if this target is not reached, then they will get back their funds after the expiration date. The start-up wants to ensure that contributions cannot be retracted before time t, or once v₿ have been gathered.

We implement this use case without covenants, but just constraining the val field of the redeeming transaction. To fund the campaign, a contributor A_i publishes the transaction T_i in Figure 4.3 (left), which uses the following script:

$$\mathit{CF} \;=\; \begin{array}{l} (\mathsf{versig}(\mathsf{Z}, \mathsf{rtxo}(1).\mathsf{wit}) \text{ and } \mathsf{rtxo}(1).\mathsf{val} \geq v) \;\; \text{or} \\ \mathsf{absAfter}\ t : \mathsf{versig}(\mathsf{A}_i, \mathsf{rtxo}(1).\mathsf{wit}) \end{array}$$

This script is a disjunction between two conditions. The first condition allows Z to redeem the bitcoins deposited in this output, provided that the output at index 1 of the redeeming transaction pays at least v₿ (note that this constraint, rendered as $\mathsf{rtxo}(1).\mathsf{val} \geq v$, is not expressible in pure Bitcoin). The second condition allows A_i to get back her contribution after the expiration date t.

Once contributors have deposited enough funds (i.e., there are n transactions $\mathsf{T}_1, \ldots, \mathsf{T}_n$ with $v' = v_1 + \cdots v_n \geq v$), Z can get v'₿ by appending T_Z to the blockchain. Note that, compared to the assurance contract in the Bitcoin wiki [86], ours offers more protection to the start-up. Indeed, while in [86] any contributor can retract her funds at any time, this is not possible here until time t.

T_0
in: $\mathsf{U}_{\mathsf{A}}^{1\text{B}}$
wit: $sig_{\mathsf{A}}(\mathsf{T}_0)$
out: $\{\mathsf{arg} : \mathsf{A},\ \mathsf{scr} : \mathit{NFT},\ \mathsf{val} : 1\text{B}\}$

T_1
in: $(\mathsf{T}_0, 1)$
wit: $sig_{\mathsf{A}}(\mathsf{T}_1)$
out: $\{\mathsf{arg} : \mathsf{B},\ \mathsf{scr} : \mathit{NFT},\ \mathsf{val} : 1\text{B}\}$

Figure 4.4: A *creates a token with* T_0*, and transfers it to* B *with* T_1*.*

T_2
in: $(\mathsf{T}_{\mathsf{A}}, 1)\ (\mathsf{T}'_{\mathsf{A}}, 1)$
wit: $sig_{\mathsf{A}}(\mathsf{T}_2)\ sig_{\mathsf{A}}(\mathsf{T}_2)$
out(1): $\{\mathsf{arg} : \mathsf{A},\ \mathsf{scr} : \mathit{NFT},\ \mathsf{val} : 1\text{B}\}$
out(2): $\{\mathsf{arg} : \varepsilon,\ \mathsf{scr} : \mathsf{versig}(\mathsf{A}, \mathsf{rtxo}(1).\mathsf{wit}),\ \mathsf{val} : 1\text{B}\}$

Figure 4.5: A *exploits the flaw to destroy a token, redeeming its value.*

4.2.2 Non-fungible tokens

A non-fungible token represents the ownership of a physical or logical asset, which can be transferred between users. Unlike fungible tokens (e.g., ERC-20 tokens in Ethereum [106]), where each token unit is interchangeable with every other unit, non-fungible ones have unique identities. Further, they do not support split and join operations, unlike fungible tokens.

We start by implementing a subtly flawed version of the non-fungible token. Consider the transactions in Figure 4.4, which use the following script:

$$\mathit{NFT} \;=\; \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}, \mathsf{rtxo}(1).\mathsf{wit}) \text{ and } \mathsf{verrec}(1) \text{ and } \mathsf{rtxo}(1).\mathsf{val} = 1$$

User A mints a token by depositing 1B in T_0: to declare her ownership over the token, she sets $\mathsf{out}(1).\mathsf{arg}$ to her public key. To transfer the token to B, A appends the transaction T_1, setting its $\mathsf{out}(1).\mathsf{arg}$ to B's public key.

To spend T_0, the transaction T_1 must satisfy the conditions specified by the script *NFT*: (i) the wit field must contain the signature of the current owner; (ii) the script at index 1 must be equal to that at the same index in T_0; (iii) the output at index 1 must have 1B value, to preserve the integrity of the token. Once T_1 is on the blockchain, B can transfer the token to another user, by appending a transaction which redeems T_1.

The script *NFT* has a design flaw, already spotted in [63]: we show how A can exploit this flaw in Figure 4.5. Suppose we have two unspent transactions: T_{A} and T'_{A}, both representing a token owned by A (in their first and only output). The transaction T_2 can spend both of them, since it complies with all the validity conditions: indeed, *NFT* only constrains the script in the first output of the redeeming transaction, while the other outputs are only subject to the standard validity conditions (in particular, that the sum of their values does not exceed the value in input). Actually, T_2 destroys one of the two tokens, and removes the covenant from the other one.

To solve this issue, we can amend the NFT script as follows:

$$\begin{aligned} NFT' \; &= \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}\mathsf{outidx}, \mathsf{rtxo}(1).\mathsf{wit}) \text{ and} \\ &\quad \mathsf{verrec}(\mathsf{inidx}) \text{ and } \mathsf{rtxo}(\mathsf{inidx}).\mathsf{val} = 1 \end{aligned}$$

The amended script correctly handles the case of a transaction which uses different outputs to store different tokens. NFT' uses $\mathsf{ctxo}.\mathsf{arg}\mathsf{outidx}$, instead of $\mathsf{ctxo}.\mathsf{arg}1$ in NFT, to ensure that, when redeeming a given output, the signature of the owner of the token at *that* output is checked. Further, NFT' uses $\mathsf{verrec}(\mathsf{inidx})$, instead of $\mathsf{verrec}(1)$ in NFT, to ensure that the covenant is propagated exactly to the transaction output which is redeeming that token (i.e., the one at index inidx). Notice that the amendment would make T_2 invalid: indeed, the script in $\mathsf{T}'_{\mathsf{A}}.\mathsf{out}(1)$ would evaluate to false:

$$\begin{aligned} [\![NFT']\!]_{\mathsf{T}_2,2} &= [\![\mathsf{verrec}(\mathsf{inidx})]\!]_{\mathsf{T}_2,2} \;\wedge\; \cdots \\ &= (\mathsf{T}_2, [\![\mathsf{inidx}]\!]_{\mathsf{T}_2,2}).\mathsf{scr} \equiv \mathsf{T}_2.\mathsf{in}(2).\mathsf{scr} \;\wedge\; \cdots \\ &= (\mathsf{T}_2, 2).\mathsf{scr} \equiv (\mathsf{T}'_{\mathsf{A}}, 1).\mathsf{scr} \;\wedge\; \cdots \\ &= \mathsf{versig}(\mathsf{A}, \mathsf{rtxo}(1).\mathsf{wit}) \equiv NFT' \;\wedge\; \cdots \\ &= \mathit{false} \end{aligned}$$

An alternative patch, originally proposed in [63], is to add a unique identifier id to each token, e.g. by amending the NFT script as follows:

$$NFT \text{ and } id = id$$

This allows to mint distinguishable tokens. For instance, if the tokens in T_{A} and T'_{A} are distinguishable, T_2 cannot redeem both of them.

4.2.3 Vaults

Transaction outputs are usually secured by cryptographic keys (e.g. through the script $\mathsf{versig}(pk_{\mathsf{A}}, \mathsf{rtx}.\mathsf{wit})$). Whoever knows the corresponding private key (e.g., sk_{A}) can redeem such an output: in case of key theft, the legitimate owner is left without defence. Vault transactions, introduced in [63], are a technique to mitigate this issue, by allowing the legitimate owner to abort the transfer.

T_V
in: $\mathsf{U}_{\mathsf{A}}^{1₿}$
wit: $\cdots$
out: $\{\mathsf{scr} : V, \mathsf{val} : 1₿\}$

T_S
in: T_V
wit: $sig_{\mathsf{A}}(\mathsf{T}_S)$
out: $\{\mathsf{arg} : \mathsf{B}, \mathsf{scr} : S, \mathsf{val} : 1₿\}$

T
in: T_S
wit: $sig_{\mathsf{B}}(\mathsf{T})$
out: $\{\mathsf{scr} : \mathsf{versig}(\mathsf{B}, \mathsf{rtxo}(1).\mathsf{wit}), \mathsf{val} : 1₿)$
relLock: t

Figure 4.6: *Transactions for the basic vault.*

To create a vault, A deposits 1₿ in a transaction T_V with the script V:

$$V = \mathsf{versig}(\mathsf{A}, \mathsf{rtxo}(1).\mathsf{wit}) \text{ and } \mathsf{verscr}(1, S)$$

$$S = \big(\mathsf{relAfter}\ t : \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}, \mathsf{rtxo}(1).\mathsf{wit})\big) \text{ or } \mathsf{versig}(\mathsf{Ar}, \mathsf{rtxo}(1).\mathsf{wit})$$

The transaction T_V can be redeemed with the signature of A, but only by a *de-vaulting* transaction like T_S in Figure 4.6, which uses the script S. The output of the de-vaulting transaction T_S can be spent by the user set in its arg field, but only after a certain time t (e.g., by the transaction T in Figure 4.6). Before time t, A can cancel the transfer by spending T_S with her recovery key Ar.

A recursive vault The vault in Figure 4.6 has a potential issue, in that the recovery key may also be subject to theft. Although this issue is mitigated by hardware wallets (and by the infrequent need to interact with the recovery key), the vault modelled above does not discourage any attempt at stealing the key.

The issue can be solved by using a recursive covenant in the vault script R:

```
if $\mathsf{ctxo}.\mathsf{arg}.1 = 0$                                          // current state: vault
        then $\mathsf{versig}(\mathsf{A}, \mathsf{rtxo}(1).\mathsf{wit})$ and $\mathsf{verrec}(1)$ and
        $\mathsf{rtxo}(1).\mathsf{arg}.1 = 1$                                   // next state: de-vaulting
else
    $(\mathsf{relAfter}\ t : \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}.2, \mathsf{rtxo}(1).\mathsf{wit}))$ or     // current state: de-vaulting
        $\mathsf{versig}(\mathsf{Ar}, \mathsf{rtxo}(1).\mathsf{wit})$ and $\mathsf{verrec}(1)$ and
        $\mathsf{rtxo}(1).\mathsf{arg}.1 = 0$                                   // next state: vault
```

In this version of the contract, the vault and de-vaulting transactions (in Figure 4.7) have the same script. The first element of the arg sequence

T_V
in: $\mathsf{U}_{\mathsf{A}}^{1\text{₿}}$
wit: $\cdots$
out: $\{\mathsf{arg} : 0, \mathsf{scr} : R, \mathsf{val} : 1\text{₿}\}$

T_S
in: T_V
wit: $sig_{\mathsf{A}}(\mathsf{T}_S)$
out: $\{\mathsf{arg} : 1\mathsf{B}, \mathsf{scr} : R, \mathsf{val} : 1\text{₿}\}$

T_R
in: T_S
wit: $sig_{\mathsf{Ar}}(\mathsf{T}_R)$
out: $\{\mathsf{arg} : 0, \mathsf{scr} : R, \mathsf{val} : 1\text{₿}\}$

Figure 4.7: *Transactions for the recursive vault.*

encodes the contract state (0 models the vault state, and 1 the de-vaulting state), while the second element is the user who can receive the bitcoin deposited in the vault. The recovery key Ar can only be used to append the re-vaulting transaction T_R, locking again the bitcoin into the vault.

Note that key theft becomes ineffective: indeed, even if both keys are stolen, the thief cannot take control of the bitcoin in the vault, as A can keep re-vaulting.

4.2.4 A pyramid scheme

Ponzi schemes are financial frauds which lure users under the promise of high profits, but which actually repay them only with the investments of new users. A pyramid scheme is a Ponzi scheme where the scheme creator recruits other investors, who in turn recruit other ones, and so on. Unlike in Ethereum, where several Ponzi schemes have been implemented as smart contracts [14, 34], the limited expressive power of Bitcoin contract only allows for off-chain schemes [74].

We design the first "smart" pyramid scheme in Bitcoin using the transactions in Figure 4.8, where:

$$\begin{aligned} P &= \mathsf{verscr}(1, X) \text{ and } \mathsf{rtxo}(1).\mathsf{arg} = \mathsf{ctxo}.\mathsf{arg} \text{ and } \mathsf{rtxo}(1).\mathsf{val} = 2 \\ &\quad \text{and } \mathsf{verrec}(2) \text{ and } \mathsf{verrec}(3) \\ X &= \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}, \mathsf{rtxo}(1).\mathsf{wit}) \end{aligned}$$

To start the scheme, a user A_0 deposits 1₿ in the transaction T_0 (we burn this bitcoin for uniformity, so that each user earns at most 1₿ from the scheme). To make a profit, A_0 must convince other two users, say A_1 and A_2, to join the scheme. This requires the cooperation of A_1 and A_2 to publish a transaction which redeems T_0. The script P ensures that this redeeming transaction has the form of T_1 in Figure 4.8, i.e. $\mathsf{out}(1)$

T_0
in: $\mathsf{U}^{1฿}_{\mathsf{A}_0}$
wit: $sig_{\mathsf{A}_0}(\mathsf{T}_0)$
out: $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : P, \mathsf{val} : 0฿\}$

T_1
in: $\mathsf{T}_0\ \mathsf{U}^{1฿}_{\mathsf{A}_1}\ \mathsf{U}^{1฿}_{\mathsf{A}_2}$
wit: $\perp\ sig_{\mathsf{A}_1}(\mathsf{T}_1)\ sig_{\mathsf{A}_2}(\mathsf{T}_1)$
out(1): $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : X, \mathsf{val} : 2฿\}$
out(2): $\{\mathsf{arg} : \mathsf{A}_1, \mathsf{scr} : P, \mathsf{val} : 0฿\}$
out(3): $\{\mathsf{arg} : \mathsf{A}_2, \mathsf{scr} : P, \mathsf{val} : 0฿\}$

Figure 4.8: *Transactions for the pyramid scheme.*

transfers 2฿ to A_0, while the scripts in $\mathsf{out}(2)$ and $\mathsf{out}(3)$ ensure that the same behaviour is recursively applied to A_1 and A_2.

Overall, the contract ensures that, as long as new users join the scheme, each one earns 1฿. Of course, as in any Ponzi scheme, at a certain point it will no longer be possible to find new users, so those at the leaves of the transaction tree will just lose their investment.

4.2.5 King of the Ether Throne

King of the Ether Throne [111] is an Ethereum contract, which has been popular for a while around 2016, until a bug caused its funds to be frozen. The contract is initiated by a user, who pays an entry fee v_0 to become the "king". Another user can usurp the throne by paying $v_1 = 1.5v_0$ fee to the old king, and so on until new usurpers are available. Of course this leads to an exponential growth of the fee needed to become king, so subsequent versions of the contract introduced mechanisms to make the current king die if not ousted within a certain time. Although the logic to distribute money substantially differs from that in Section 4.2.4, this is still an instance of Ponzi scheme, since investors are only paid with the funds paid by later investors.

We implement the original version of the contract, fixing the multiplier to 2 instead of 1.5, since Bitcoin scripts do not support multiplication. The contract uses the transactions in Figure 4.9 for the first two kings, A_0 and A_1, where:

$$\begin{aligned} K &= \mathsf{verrec}(1)\ \text{ and }\ \mathsf{rtxo}(2).\mathsf{arg} = \mathsf{ctxo}.\mathsf{arg}\ \text{ and} \\ &\quad \mathsf{rtxo}(2).\mathsf{val} \geq \mathsf{ctxo}.\mathsf{val} + \mathsf{ctxo}.\mathsf{val}\ \text{ and } \mathsf{verscr}(2, X) \\ X &= \mathsf{versig}(\mathsf{ctxo}.\mathsf{arg}, \mathsf{rtxo}(1).\mathsf{wit}) \end{aligned}$$

We use the arg field in $\mathsf{out}(1)$ to record the new king, and that in $\mathsf{out}(2)$ for the old one. The clause $\mathsf{rtxo}(2).\mathsf{arg} = \mathsf{ctxo}.\mathsf{arg}$ in K preserves the old king in the redeeming transaction. The clause $\mathsf{rtxo}(2).\mathsf{val} \geq$

T_0
in: $\mathsf{U}^{v_0 Ƀ}_{\mathsf{A}_0}$
wit: $sig_{\mathsf{A}_0}(\mathsf{T}_0)$
$\mathsf{out}(1)$: $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : K, \mathsf{val} : 0Ƀ\}$
$\mathsf{out}(2)$: $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : \mathsf{versig}(\mathsf{A}_0, \mathsf{rtxo}(1).\mathsf{wit}), \mathsf{val} : v_0Ƀ\}$

T_1
in: $(\mathsf{T}_0, 1)$ $\mathsf{U}^{v_1 Ƀ}_{\mathsf{A}_1}$
wit: $\perp$ $sig_{\mathsf{A}_1}(\mathsf{T}_1)$
$\mathsf{out}(1)$: $\{\mathsf{arg} : \mathsf{A}_1, \mathsf{scr} : K, \mathsf{val} : 0Ƀ\}$
$\mathsf{out}(2)$: $\{\mathsf{arg} : \mathsf{A}_0, \mathsf{scr} : X, \mathsf{val} : v_1Ƀ\}$

Figure 4.9: *Transactions for King of the Ether Throne.*

$\mathsf{ctxo}.\mathsf{val} + \mathsf{ctxo}.\mathsf{val}$ ensures that his compensation is twice the value he paid. Finally, verscr guarantees that the old king can redeem his compensation via $\mathsf{out}(2)$.

4.3 Using covenants in high-level contract languages

As witnessed by the use cases in Section 4.2, crafting a contract at the level of Bitcoin transactions can be complex and error-prone. To simplify this task, the work described in [22] has introduced a high-level contract language, called BitML, with a secure compiler to pure Bitcoin transactions. BitML has primitives to `withdraw` funds from a contract, to `split` a contract (and its funds) into subcontracts, to request the authorization from a participant A before proceeding with a subcontract C (written $\mathsf{A} : C$), to postpone the execution of C after a given time t (written `after` $t : C$), to `reveal` committed secrets, and to branch between two contracts (written $C + C'$). A recent paper [20] extends BitML with a new primitive that allows participants to (consensually) renegotiate a contract, still keeping the ability to compile to pure Bitcoin.

Despite the variety of use cases shown in [10, 15], BitML has known expressiveness limits, given by the requirement to have pure Bitcoin as its compilation target. For instance, BitML cannot specify recursive contracts (just as pure Bitcoin cannot), unless all participants agree to perform the recursive call [20]. In this section we discuss how to improve the expressiveness of BitML, assuming to use Bitcoin with covenants as compilation target. We illustrate our point by a couple of examples, postponing the formal treatment of this extended BitML and of its secure compilation to future work.

Covenants allow us to extend BitML with the construct:

$$?\boldsymbol{x}\ \texttt{if}\ b.\ \texttt{X}\langle\boldsymbol{x}\rangle$$

Intuitively, the prefix $?\boldsymbol{x}\ \texttt{if}\ b$ can be fired whenever a participant provides a sequence of arguments $\boldsymbol{x}$ and makes the predicate b true. Once the prefix is fired, the contract proceeds as the continuation $\texttt{X}\langle\boldsymbol{x}\rangle$, which will reduce according to the equation defining $\texttt{X}$.

Using this construct, we can model the "King of the Ether Throne" contract of Section 4.2.5 (started by A with an investment of 1₿) as $\texttt{X}\langle\mathsf{A}, 1\rangle$, where:

$$\begin{aligned}
\texttt{X}\langle\mathsf{a}, v\rangle &= ?\mathsf{b}\ \texttt{if}\ \mathsf{val} \geq 2v.\ \texttt{Y}\langle\mathsf{a}, \mathsf{b}, \mathsf{val}\rangle \\
\texttt{Y}\langle\mathsf{a}, \mathsf{b}, v\rangle &= \texttt{split}\ \big(0 \to \texttt{X}\langle\mathsf{b}, v\rangle \mid v \to \texttt{withdraw}\ \mathsf{a}\big)
\end{aligned}$$

The contract $\texttt{X}\langle\mathsf{a}, v\rangle$ models a state where a is the current king, and v is his investment. The guard $\mathsf{val} \geq 2v$ becomes true when some participant injects funds into the contract, making its value (val) greater than $2v$. This participant can choose the value for b, i.e. the new king. The contract proceeds as $\texttt{Y}\langle\mathsf{a}, \mathsf{b}, \mathsf{val}\rangle$, which has two parallel branches. The first branch makes $\mathsf{val}\,\text{₿}$ available to the old king; the second branch has zero value, and it reboots the game, recording the new king b and his investment.

A possible computation of A starting the scheme with 1₿ is the following, where we represent a contract C storing $v\text{₿}$ as a term $\langle C, v\text{₿}\rangle$:

$$\begin{array}{ll}
\langle\texttt{X}\langle\mathsf{A}, -\rangle, 1\text{₿}\rangle & \\
\to \langle\texttt{Y}\langle\mathsf{A}, \mathsf{B}, 2\rangle, 2\text{₿}\rangle & (\mathsf{B}\text{ pays }2\text{₿ fee}) \\
\to \langle\texttt{X}\langle\mathsf{B}, 2\rangle, 0\text{₿}\rangle \mid \langle\texttt{withdraw}\ \mathsf{A}, 2\text{₿}\rangle & (\text{contract splits}) \\
\to \langle\texttt{X}\langle\mathsf{B}, 2\rangle, 0\text{₿}\rangle \mid \langle\mathsf{A}, 2\text{₿}\rangle & (\mathsf{A}\text{ redeems }2\text{₿}) \\
\to \langle\texttt{Y}\langle\mathsf{B}, \mathsf{C}, 4\rangle, 4\text{₿}\rangle \mid \langle\mathsf{A}, 2\text{₿}\rangle & (\mathsf{C}\text{ pays }4\text{₿ fee}) \\
\to \langle\texttt{X}\langle\mathsf{C}, 4\rangle, 0\text{₿}\rangle \mid \langle\texttt{withdraw}\ \mathsf{B}, 4\text{₿}\rangle \mid \langle\mathsf{A}, 2\text{₿}\rangle & (\text{contract splits}) \\
\to \langle\texttt{X}\langle\mathsf{C}, 4\rangle, 0\text{₿}\rangle \mid \langle\mathsf{B}, 4\text{₿}\rangle \mid \langle\mathsf{A}, 2\text{₿}\rangle & (\mathsf{B}\text{ redeems }4\text{₿})
\end{array}$$

Executing a step of the BitML contract corresponds, in Bitcoin, to appending a transaction containing in $\mathsf{out}(1)$ the script in Figure 4.10. The script implements a state machine, using $\mathsf{arg}.1$ to record the current state, and the other parts of arg for the old king, the new king, and v. The $\mathsf{verrec}(1)$ at line 8 preserves the script in $\mathsf{out}(1)$. To pay the old king, we use the verscr at line 20, which constrains the script in $\mathsf{out}(2)$ of the transaction corresponding to the BitML state $\langle\texttt{X}\langle\mathsf{b}, v\rangle, 0\text{₿}\rangle \mid \langle\texttt{withdraw}\ \mathsf{a}, v\text{₿}\rangle$.

```
def arg.1 = q      // state    0 = <X(A,-),1>
                   // state    1 = <Y(a,b,v),v>
                   // state    2 = <X(b,v),0> | <withdraw a,v>
def arg.2 = oldK   // old King
def arg.3 = newK   // new king
def arg.4 = v      // paid fee

verrec(1) and                                 // out(1) preserves covenant
if ctxo(1).q = 0 then                         // state 0
      rtxo(1).q = 1                           // state transition 0 -> 1
  and rtxo(1).oldK = ctxo(1).newK             // usurp the throne
  and rtxo(1).val >= ctxo(1).val
                     + ctxo(1).val            // fee at least doubled
  and rtxo(1).v = rtxo(1).val                 // instantiate v
else if ctxo(1).q = 1 then                    // state 1
      rtxo(1).q = 2                           // state transition 1 -> 2
  and rtxo(1).newK = ctxo(1).newK             // preserve new king
  and rtxo(1).v = ctxo(1).v                   // preserve v
  and rtxo(1).val = 0                         // reset value in out(1)

  and rtxo(2).oldK = ctxo(1).oldK             // set old king
  and verscr(2,
        versig(ctxo(2).oldK,rtx.wit))         // covenant to pay old king
  and rtxo(2).val = ctxo(1).val               // preserve value in out(2)
else if ctxo(1).q = 2 then                    // state 2
      rtxo(1).q = 1                           // state transition 2 -> 1
  and rtxo(1).oldK = ctxo(1).newK             // usurp the throne
  and rtxo(1).val >= ctxo(1).v
                     + ctxo(1).v              // fee at least doubled
  and rtxo(1).v = rtxo(1).val                 // update v
```

Figure 4.10: *Script for King of the Ether Throne, obtained by compiling BitML.*

We now apply our extended BitML to specify a more challenging use case, i.e. a recursive coin-flipping game where two players A and B repeatedly flip coins, and the one who wins two consecutive flips takes the pot. The precondition to stipulate the contract requires each player to deposit 1₿ as a bet. The game first makes each player commit to a secret, using a timed-commitment protocol [29]. The secrets are then revealed, and the winner of a flip is determined as a function of the two secrets. The game starts another flip if the current winner is different from that of the previous flip, otherwise the pot is transferred to the winner.

We model the recursive coin-flipping game as the (extended) BitML contract $\mathtt{X}_{\mathsf{A}}\langle \mathsf{C} \rangle$, where $\mathsf{C} \neq \mathsf{A}, \mathsf{B}$, using the following defining equations:

$$\begin{aligned}
\mathtt{X}_{\mathsf{A}}\langle \mathsf{w} \rangle &= \mathsf{A} : ?h_{\mathsf{A}} \,.\, \mathtt{X}_{\mathsf{B}}\langle \mathsf{w}, h_{\mathsf{A}} \rangle \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B} \\
\mathtt{X}_{\mathsf{B}}\langle \mathsf{w}, h_{\mathsf{A}} \rangle &= \mathsf{B} : ?h_{\mathsf{B}} \,.\, \mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle \;+\; \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A} \\
\mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle &= ?s_{\mathsf{A}}\ \mathtt{if}\ H(s_{\mathsf{A}}) = h_{\mathsf{A}} \,.\, \mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle \\
&\quad + \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{B} \\
\mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle &= ?s_{\mathsf{B}}\ \mathtt{if}\ H(s_{\mathsf{B}}) = h_{\mathsf{B}}\ \mathsf{and}\ 0 \leq s_{\mathsf{B}} \leq 1 \,.\, \mathtt{W}\langle \mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}} \rangle \\
&\quad + \mathtt{afterRel}\, t : \mathtt{withdraw}\ \mathsf{A} \\
\mathtt{W}\langle \mathsf{w}, s_{\mathsf{A}}, s_{\mathsf{B}} \rangle &= \ \ \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} = \mathsf{A} : \mathtt{withdraw}\ \mathsf{A} \qquad // \ \mathsf{A}\text{ won twice} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} = s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} \neq \mathsf{A} : \mathtt{X}_{\mathsf{A}}\langle \mathsf{A} \rangle \qquad // \ \mathsf{A}\text{ won last flip} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} = \mathsf{B} : \mathtt{withdraw}\ \mathsf{B} \qquad // \ \mathsf{B}\text{ won twice} \\
&\quad + \mathtt{if}\ s_{\mathsf{A}} \neq s_{\mathsf{B}}\ \mathsf{and}\ \mathsf{w} \neq \mathsf{B} : \mathtt{X}_{\mathsf{A}}\langle \mathsf{B} \rangle \qquad // \ \mathsf{B}\text{ won last flip}
\end{aligned}$$

The contract $\mathtt{X}_{\mathsf{A}}\langle \mathsf{w} \rangle$ models a state where w is the last winner, and A must commit to her secret. To do that, A must authorize an input h_{A}, which represents the hash of her secret. If A does not commit within t, then the pot can be redeemed by B as a compensation (here, the primitive $\mathtt{afterRel}\, t : C$ models a relative timeout). Similarly, $\mathtt{X}_{\mathsf{B}}\langle \mathsf{w} \rangle$ models B's turn to commit. In $\mathtt{Y}_{\mathsf{A}}\langle \mathsf{w}, h_{\mathsf{A}}, h_{\mathsf{B}} \rangle$, A must reveal her secret s_{A}, or otherwise lose her deposit. The contract $\mathtt{Y}_{\mathsf{B}}\langle \mathsf{w}, s_{\mathsf{A}}, h_{\mathsf{B}} \rangle$ is the same for B, except that here we additionally check that B's secret is either 0 or 1 (this is needed to ensure fairness, as in the two-player lottery in [22]). The flip winner is A if the secrets of A and B are equal, otherwise it is B. If the winner is the same as the previous round, the winner can withdraw the pot, otherwise the game restarts, recording the last winner.

This coin flipping game is fair, i.e. the expected payoff of a *rational* player is always non-negative, notwithstanding the behaviour of the other player.

4.4 Implementing neighbourhood covenants

To extend Bitcoin with neighborhood covenants, only small changes to the script language are needed. Currently, scripts can only access the redeeming transaction, and only for signature verification. To enable covenants, scripts need to access the fields of the redeeming transaction, and those of the parent and the *sibling* transaction outputs. First, the UTXO data structure must be extended to record the parents of unspent outputs. Full nodes could simply access these transactions from their identifiers. Lightweight nodes, i.e. Bitcoin clients with limited resources that store the UTXO instead of the whole blockchain, must also store parents beside the

UTXO; when all the children of a transaction are spent, the parent can be deleted.

To implement the covenants verrec and verscr, the Bitcoin script language must be extended with new opcodes. A former covenant-enabling opcode is the CheckOutputVerify of [63], which uses placeholders to represent variable parts of the script (e.g., versig(<*pubKey*>, rtx.wit)). However, its implementation requires string substitutions at run-time to insert the wanted values in the script before checking script equality. Instead, our neighbourhood covenants can be implemented more efficiently. In our covenants, we can use exactly the same script along a chain of transactions, relying on the arg sequence to record state updates. The script can access the state through the operator o.arg, which require suitable opcodes. Since the script is fixed, nodes do not have to perform string substitutions, and checking script equality can be efficiently performed by comparing their hashes.

Adding the arg field to outputs does not require to alter the structure of pure Bitcoin transactions. Indeed, the arg values can be stored at the beginning of the script as push operations on the alternative stack, and copied to the main stack when the script refers to them, using the same technique used in [81]. When hashing the scripts for comparison, we discard these arg values: this just requires to skip the prefix of the script comprising all the push to the alternative stack.

Chapter 5

Bitcoin Smart Contracts as Endpoint Protocols

Albeit the primary usage of Bitcoin is to exchange currency, its blockchain and consensus mechanism can also be exploited to securely execute some forms of *smart contracts*. These are agreements among mutually distrusting parties, which can be automatically enforced without resorting to a trusted intermediary. Over the last few years a variety of smart contracts for Bitcoin have been proposed, both by the academic community and by that of developers. However, the heterogeneity in their treatment, the informal (often incomplete or imprecise) descriptions, and the use of poorly documented Bitcoin features, pose obstacles to the research. In this chapter we present a comprehensive survey of smart contracts on Bitcoin, in a uniform framework. Our treatment is based on a new formal specification language for smart contracts, which also helps us to highlight some subtleties in existing informal descriptions, making a step towards automatic verification. We discuss some obstacles to the diffusion of smart contracts on Bitcoin, and we identify the most promising open research challenges.

5.1 Modelling Bitcoin contracts

In this section we introduce a formal model of the behaviour of the participants in a contract, building upon the model of Bitcoin transactions in Chapter 3.

We start by formalising a simple language of expressions, which represent both the messages sent over the network, and the values used in internal computations made by the participants. Hereafter, we assume a

$$[\![\nu]\!] = \nu \qquad [\![\mathsf{sig}_k^{\mu,i}(T)]\!] = \mathsf{sig}_k^{\mu,i}([\![T]\!])$$

$$[\![\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T, i)]\!] = \mathsf{ver}_{\boldsymbol{k}}([\![\boldsymbol{E}]\!], [\![T]\!], i)$$

$$[\![T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\}]\!] = [\![T]\!]\{\mathsf{f}(i) \mapsto [\![\boldsymbol{E}]\!]\} \qquad [\![(E, E')]\!] = ([\![E]\!], [\![E']\!])$$

$$[\![E \circ E']\!] = [\![E]\!] \circ [\![E']\!] \quad \text{for } \circ \in \{\ \mathsf{and}\,,\ \mathsf{or}\,, +, \ldots\} \qquad [\![\mathsf{not}\ E]\!] = \neg[\![E]\!]$$

$$[\![\boldsymbol{E}]\!] = [\![E_1]\!] \cdots [\![E_n]\!] \quad \text{if } \boldsymbol{E} = E_1 \cdots E_n$$

Figure 5.1: *Semantics of contract expressions.*

set Var of *variables*, and we define the set Val of *values* comprising constants $k \in \mathbb{Z}$, signatures σ, scripts $\lambda\boldsymbol{z}.e$, transactions T, and currency values v.

Definition 5.1 (Contract expressions)**.** *We define* contract expressions *through the following syntax:*

$$\begin{array}{rcll}
E, T & ::= & \nu & \textit{value } (\nu \in \mathsf{Val}) \\
& \mid & x & \textit{variable } (x \in \mathsf{Var}) \\
& \mid & \mathsf{sig}_k^{\mu,i}(T) & \textit{signature } (\mu \textit{ signature modifier}) \\
& \mid & \mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T, i) & \textit{(multi) signature verification} \\
& \mid & T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\} & \textit{transaction field update} \\
& \mid & (E, E) & \textit{pair} \\
& \mid & E \ \mathsf{and}\ E \mid E\ \mathsf{or}\ E \mid \mathsf{not}\ E & \textit{logical expressions} \\
& \mid & E + E \mid \cdots & \textit{arithmetic expressions}
\end{array}$$

where $\boldsymbol{E}$ denotes a finite sequence of expressions (i.e., $\boldsymbol{E} = E_1 \cdots E_n$). We define the function $[\![\cdot]\!]$ from (variable-free) contract expressions to values in Figure 5.1. As a notational shorthand, we omit the index i in sig *(resp.* versig*) when the signed (resp. verified) transactions have a single input.*

Intuitively, when T evaluates to a transaction T, the expression $T\{\mathsf{f}(i) \mapsto \boldsymbol{E}\}$ represents the transaction obtained from T by substituting the field $\mathsf{f}(i)$ with the sequence of values obtained by evaluating $\boldsymbol{E}$. For instance, $\mathsf{T}\{\mathsf{wit}(1) \mapsto \sigma\}$ denotes the transaction obtained from T by replacing the witness at index 1 with the signature σ. Further, $\mathsf{sig}_k^{\mu,i}(T)$ evaluates to the signature of the transaction represented by T, and $\mathsf{versig}_{\boldsymbol{k}}(\boldsymbol{E}, T)$ represents the m-of-n multi-signature verification of the transaction represented by T. Both for the signing and verification, the parameter i represents the index where the signature will be used. We assume a simple type system that rules out ill-formed expressions, like e.g. $k\{\mathsf{wit}(1) \mapsto \mathsf{T}\}$.

We formalise the behaviour of a participant as an *endpoint protocol*, i.e. a process where the participant can perform the following actions: (i) send/receive messages to/from other participants; (ii) put a transaction on the ledger; (iii) wait until some transactions appear on the blockchain; (iv) do some internal computation. Note that the last kind of operation allows a participant to craft a transaction before putting it on the blockchain, e.g. setting the wit field to her signature, and later on adding the signature received from another participant.

Definition 5.2 (Endpoint protocols)**.** *Assume a set of* participants *(named* A*,* B*,* C*, . . .). We define* prefixes π*, and* protocols $P, Q, R, \ldots$ *as follows:*

π	$::= \mathsf{A}\,!\,\boldsymbol{E}$	*send messages to* A
	$\mid \mathsf{A}\,?\,\boldsymbol{x}$	*receive messages from* A
	$\mid \mathsf{put}\ T$	*append transaction* T *to the blockchain*
	$\mid \mathsf{ask}\ \boldsymbol{T}\ \mathsf{as}\ \boldsymbol{x}$	*wait until all transactions in* $\boldsymbol{T}$ *are on the blockchain*
	$\mid \mathsf{check}\ E$	*test condition*
P	$::= \sum_{i \in I} \pi_i \,.\, P_i$	*guarded choice (I finite set)*
	$\mid P \mid P$	*parallel composition*
	$\mid \mathrm{X}(\boldsymbol{E})$	*named process*

We assume that each name X *has a unique defining equation* $\mathrm{X}(\boldsymbol{x}) = P$ *where the free variables in* P *are included in* $\boldsymbol{x}$*. We use the following syntactic sugar:*

- $\tau \triangleq \mathsf{check}\ \mathit{true}$*, the* internal action*;*
- $\mathbf{0} \triangleq \sum_{\emptyset} P$*, the* terminated *protocol (as usual, we omit trailing* $\mathbf{0}$*s);*
- $\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleq \mathsf{check}\ E\ .\, P + \mathsf{check}\ \mathsf{not}\ E\ .\, Q$*;*
- $\pi_1.Q_1 + P \triangleq \sum_{i \in I \cup \{1\}} \pi_i.Q_i$*, provided that* $P = \sum_{i \in I} \pi_i.Q_i$ *and* $1 \notin I$*;*
- $\mathsf{let}\ x = E\ \mathsf{in}\ P \triangleq P\{E/x\}$*, i.e.* P *where* x *is replaced by* E*.*

The behaviour of protocols is defined in terms of a LTS between *sys-*

$$\mathsf{A}[\mathsf{B}\,!\,\boldsymbol{E}.\,P + R \mid Q] \mid \mathsf{B}[\mathsf{A}\,?\,\boldsymbol{x}.\,P' + R' \mid Q'] \mid S \to \mathsf{A}[P \mid Q] \mid \mathsf{B}[P'\{[\![\boldsymbol{E}]\!]/\boldsymbol{x}\} \mid Q'] \mid S \quad [\textsc{Com}]$$

$$\frac{[\![E]\!] = \mathit{true}}{\mathsf{A}[\mathsf{check}\ E\ .\,P + R \mid Q] \mid S \to \mathsf{A}[P \mid Q] \mid S}\ [\textsc{Check}]$$

$$\frac{[\![T]\!] = \mathsf{T} \qquad \mathsf{B} \rhd (\mathsf{T}, t)}{\mathsf{A}[\mathsf{put}\ T.\,P + R \mid Q] \mid S \mid (\mathsf{B}, t) \to \mathsf{A}[P \mid Q] \mid S \mid (\mathsf{B}(\mathsf{T}, t), t)}\ [\textsc{Put}]$$

$$\frac{[\![\boldsymbol{T}]\!] = \mathsf{T}_1 \cdots \mathsf{T}_n \quad \forall i \in 1..n : \mathit{match}_{\mathsf{B}}(\mathsf{T}_i) = \mathsf{T}'_i \neq \bot}{\mathsf{A}[\mathsf{ask}\ \boldsymbol{T}\ \mathsf{as}\ \boldsymbol{x}.\,P + R \mid Q] \mid S \mid (\mathsf{B}, t) \to \mathsf{A}[P\{\mathsf{T}'_1 \cdots \mathsf{T}'_n/\boldsymbol{x}\} \mid Q] \mid S \mid (\mathsf{B}, t)}\ [\textsc{Ask}]$$

$$\frac{\mathsf{X}(\boldsymbol{x}) = P \quad \mathsf{A}[P\{[\![\boldsymbol{E}]\!]/\boldsymbol{x}\} \mid Q] \mid S \to S'}{\mathsf{A}[\mathsf{X}(\boldsymbol{E}) \mid Q] \mid S \to S'}\ [\textsc{Def}] \qquad \frac{t' > 0}{S \mid (\mathsf{B}, t) \xrightarrow{t'} S' \mid (\mathsf{B}, t + t')}\ [\textsc{Delay}]$$

Figure 5.2: *Semantics of endpoint protocols.*

tems, i.e. the parallel composition of the protocols of all participants, and the blockchain.

Definition 5.3 (Semantics of protocols)**.** *A system S is a term of the form $\mathsf{A}_1[P_1] \mid \cdots \mid \mathsf{A}_n[P_n] \mid (\mathsf{B}, t)$, where (i) all the A_i are distinct; (ii) there exists a single component (B, t), representing the current state of the blockchain B, and the current time t; (iii) systems are up-to commutativity and associativity of $\mid$. We define the relation $\to$ between systems in Figure 5.2, where $\mathit{match}_{\mathsf{B}}(\mathsf{T})$ is the set of all the transactions in B that are equal to T, except for the witnesses. When writing $S \mid S'$ we intend that the conditions above are respected.*

Intuitively, a guarded choice $\sum_i \pi_i.P_i$ can behave as one of the branches P_i. A parallel composition $P \mid Q$ executes concurrently P and Q. All the rules (except the last two) specify how a protocol $(\pi.P+Q) \mid R$ evolves within a system. Rule [Com] models a message exchange between A and B: participant A sends messages $\boldsymbol{E}$, which are received by B on variables $\boldsymbol{x}$. Communication is synchronous, i.e. A is blocked until B is ready to receive. Rule [Check] allows the branch P of a sum to proceed if the condition represented by E is true. Rule [Put] allows A to append a transaction to the blockchain, provided that the update is consistent. Rule [Ask] allows the branch P of a sum to proceed only when the blockchain contains some transactions $\mathsf{T}'_1 \cdots \mathsf{T}'_n$ obtained by instantiating some $\bot$ fields in $\boldsymbol{T}$ (see Section 1.1). This form of pattern matching is crucial because the value of some fields (e.g., wit), may not be known at the time the protocol is written. When the ask prefix unblocks, the variables $\boldsymbol{x}$ in P are bound to $\mathsf{T}'_1 \cdots \mathsf{T}'_n$, so making it possible to inspect their actual

T
in: $(\mathsf{T_A}, 1)$
wit: $\bot$
out: $(\lambda \varsigma\varsigma'.\mathsf{versig}(k_\mathsf{A} k_\mathsf{B}, \varsigma\varsigma'), 1\text{B})$

$\mathsf{T'_A}$
in: $(\mathsf{T}, 1)$
wit: $\bot$
out: $(\lambda \varsigma.\mathsf{versig}(k_\mathsf{A}, \varsigma), 1\text{B})$

$\mathsf{T'_B}$
in: $(\mathsf{T}, 1)$
wit: $\bot$
out: $(\lambda \varsigma.\mathsf{versig}(k_\mathsf{B}, \varsigma), 1\text{B})$

Figure 5.3: *Transactions of the naïve escrow contract.*

fields. Rule [Def] allows a named process $\mathsf{X}(\boldsymbol{E})$ to evolve as P, assuming a defining equation $\mathsf{X}(\boldsymbol{x}) = P$. The variables $\boldsymbol{x}$ in P are substituted with the results of the evaluation of $\boldsymbol{E}$. Such defining equations can be used to specify recursive behaviours. Finally, rule [Delay] allows time to pass. To keep our presentation simple, we have not included time-constraining operators in endpoint protocols. In case one needs a finer-grained control of time, well-known techniques [64] exist to extend a process algebra like ours with these operators.

Example 5.1.1 (Naïve escrow)**.** A buyer A wants to buy an item from the seller B, but they do not trust each other. So, they would like to use a contract to ensure that B will get paid if and only if A gets her item. In a naïve attempt to realise this, they use the transactions in Figure 5.3, where we assume that $(\mathsf{T_A}, 1)$ used in $\mathsf{T.in}$, is a transaction output redeemable by A through her key k_A. The transaction T makes A deposit 1B, which can be redeemed by a transaction carrying the signatures of both A and B. The transactions $\mathsf{T'_A}$ and $\mathsf{T'_B}$ redeem T, transferring the money to A or B, respectively.

The protocols of A and B are, respectively, P_A and Q_B:

$$\begin{aligned}
P_\mathsf{A} &= \mathsf{put}\ \mathsf{T}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T})\}.\,P' \\
P' &= \tau.\mathsf{B}\,!\,\mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T'_B}) \;+\; \tau.\mathsf{B}\,?\,x.\,\mathsf{put}\ \mathsf{T'_A}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T'_A})\,x\} \\
Q_\mathsf{B} &= \mathsf{ask}\ \mathsf{T}.\,\big(\tau.\mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T'_B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T'_B})\} \;+\; \tau.\mathsf{A}\,!\,\mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T'_A})\big)
\end{aligned}$$

First, A adds her signature to T, and puts it on the blockchain. Then, she internally chooses whether to unblock the deposit for B or to request a refund. In the first case, A sends $\mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T'_B})$ to B. In the second case, she waits to receive the signature $\mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T'_A})$ from B (saving it in the variable x); afterwards, she puts $\mathsf{T'_A}$ on the blockchain (after setting wit) to redeem the deposit. The seller B waits to see T on the blockchain. Then, he chooses either to receive the signature $\mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{B})$ from A (and then redeem the payment by putting T'_B on the blockchain), or to refund A, by sending his signature $\mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})$.

This contract is not secure if either A or B are dishonest. On the one hand, a dishonest A can prevent B from redeeming the deposit, even if she had already received the item (to do that, it suffices not to send her signature, taking the rightmost branch in P'). On the other hand, a dishonest B can just avoid to send the item and the signature (taking the leftmost branch in Q_B): in this way, the deposit gets frozen. For instance, let $S = \mathsf{A}[P_\mathsf{A}] \mid \mathsf{B}[Q_\mathsf{B}] \mid (\mathbf{B}, t)$, where $\mathbf{B}$ contains T_A unredeemed. The scenario where A has never received the item, while B dishonestly attempts to receive the payment, is modelled as follows:

$$\begin{aligned}
S &\to \mathsf{A}[P'] \mid \mathsf{B}[Q_\mathsf{B}] \mid (\mathbf{B}(\mathsf{T}, t), t)\\
&\to \mathsf{A}[P'] \mid \mathsf{B}[\tau.\mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\}\ +\ \tau.\mathsf{A}\,!\,\mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{A})] \mid \cdots\\
&\to \mathsf{A}[\mathsf{B}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{A}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}'_\mathsf{A})\ x\}] \mid \mathsf{B}[\mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto x\ \mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\}] \mid \cdots
\end{aligned}$$

At this point the computation is stuck, because both A and B are waiting a message from the other participant. We will show in Section 5.2.3 how to design a secure escrow contract, with the intermediation of a trusted arbiter.

5.2 Smart Contracts

We now present a comprehensive survey of smart contracts on Bitcoin, comprising those published in the academic literature, and those found online. To this aim we exploit the model of computation introduced in Section 5.1. Remarkably, all the following contracts can be implemented by only using so-called *standard* transactions [85], e.g. via the compilation technique in [11]. This is crucial, because non-standard transactions are currently discarded by the Bitcoin network.

5.2.1 Oracle

In many concrete scenarios one would like to make the execution of a contract depend on some real-world events, e.g. results of football matches for a betting contract, or feeds of flight delays for an insurance contract. However, the evaluation of Bitcoin scripts can not depend on the environment, so in these scenarios one has to resort to a trusted third-party, or

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

T'_B
in: $(\mathsf{T}, 1)$
wit: $\perp$
out: $(\lambda \varsigma.\mathsf{versig}(k_\mathsf{B}, \varsigma), v$₿$)$

Figure 5.4: *Transactions of a contract relying on an oracle.*

oracle [87, 90], who notifies real-world events by providing signatures on certain transactions.

For example, assume that A wants to transfer v₿ to B only if a certain event, notified by an oracle O, happens. To do that, A puts on the blockchain the transaction T in Figure 5.4, which can be redeemed by a transactions carrying the signatures of both B and O. Further, A instructs the oracle to provide his signature to B upon the occurrence of the expected event.

We model the behaviour of B as the following protocol:

$$P_\mathsf{B} \;=\; \mathsf{O}\,?x.\,\mathsf{put}\;\mathsf{T}'_\mathsf{B}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}'_\mathsf{B})\; x\}$$

Here, B waits to receive the signature $\mathsf{sig}^{**}_{k_\mathsf{O}}(\mathsf{T}'_\mathsf{B})$ from O, then he puts T'_B on the blockchain (after setting its wit) to redeem T. In practice, oracles like the one needed in this contract are available as services in the Bitcoin ecosystem, eg [119].

Notice that, in case the event certified by the oracle never happens, the v₿ within T are frozen forever. To avoid this situation, one can add a time constraint to the output script of T, e.g. as in the transaction T_{bond} in Figure 5.10.

5.2.2 Crowdfunding

Assume that the curator C of a crowdfunding campaign wants to fund a venture V by collecting v₿ from a set $\{\mathsf{A}_i\}_{i \in I}$ of investors. The investors want to be guaranteed that either the required amount v₿ is reached, or they will be able to redeem their funds. To this purpose, C can employ the following contract. She starts with a canonical transaction $\mathsf{U}^{v\text{₿}}_\mathsf{V}$ (with empty in field) which has a single output of v₿ to be redeemed by V. Intuitively, each A_i can invest money in the campaign by "filling in" the in field of the $\mathsf{U}^{v\text{₿}}_\mathsf{V}$ with a transaction output under their control. To do this, A_i sends to C a transaction output (T_i, j_i), together with the signature σ_i required to redeem it. We denote with $val(\mathsf{T}_i, j_i)$ the value of such output. Notice that, since the signature σ_i has been made on $\mathsf{U}^{v\text{₿}}_\mathsf{V}$, the only valid output is the one of v₿ to be redeemed by V. Upon the reception of the message from A_i, C updates $\mathsf{U}_{\mathsf{V}}^{v₿}$: the provided output is appended to the in field, and the signature is added to the corresponding wit field. If all the outputs (T_i, j_i) are distinct (and not redeemed) and the signatures are valid, when $\sum_i \mathit{val}(\mathsf{T}_i, j_i) \geq v$ the filled transaction $\mathsf{U}_{\mathsf{V}}^{v₿}$ can be put on the blockchain. If C collects $v' > v$₿, the difference $v' - v$ goes to the miners as transaction fee.

The endpoint protocol of the curator is defined as $X(\mathsf{U}_{\mathsf{V}}^{v₿}, 1, 0)$, where:

$$X(x, n, d) \;=\; \mathsf{if}\ d < v\ \mathsf{then}\ P\ \mathsf{else\ put}\ x$$
$$P \;=\; \sum_i \mathsf{A}_i \,?\, (y, j, \sigma).\, X(x\{\mathsf{in}(n) \mapsto (y, j)\}\{\mathsf{wit}(n) \mapsto \sigma\}, n+1, d + \mathit{val}(y, j))$$

while the protocol of each investor A_i is the following:

$$P_{\mathsf{A}_i} = \mathsf{C}\,!\,(\mathsf{T}_i, j_i, \mathsf{sig}_{k_{\mathsf{A}_i}}^{1*,1}(\mathsf{U}_{\mathsf{V}}^{v₿}\{\mathsf{in}(1) \mapsto (\mathsf{T}_i, j_i)\}))$$

Note that the transactions sent by investors are not known *a priori*, so they cannot just create the final transaction and sign it. Instead, to allow C to complete the transaction $\mathsf{U}_{\mathsf{V}}^{v₿}$ without invalidating the signatures, they compute them using the modifier $1{*}_1$. In this way, only a single input is signed, and when verifying the corresponding signature, the others are neglected.

5.2.3 Escrow

In Example 5.1.1 we have discussed a naïve escrow contract, which is secure only if both the buyer A and the seller B are honest (so making the contract pointless). Rather, one would like to guarantee that, even if either A or B (or both) are dishonest, exactly one them will be able to redeem the money: in case they disagree, a trusted participant C, who plays the role of arbiter, will decide who gets the money (possibly splitting the initial deposit in two parts) [84, 90].

The output script of the transaction T in Figure 5.5 is a *2-of-3* multi-signature schema. This means that T can be redeemed either with the signatures A and B (in case they agree), or with the signature of C (with key k_{C}) and the signature of A or that of B (in case they disagree). The transaction $\mathsf{T}'_{\mathsf{AB}}(z)$ in Figure 5.5 allows the arbiter to issue a *partial* refund of z₿ to A, and of $(1-z)$₿ to B. Instead, to issue a full refund to either A or B, the arbiter signs, respectively, the transactions $\mathsf{T}'_{\mathsf{A}} = \mathsf{U}_{\mathsf{A}}^{1₿₿}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ or $\mathsf{T}'_{\mathsf{B}} = \mathsf{U}_{\mathsf{B}}^{1₿₿}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ (not shown in the figure). The protocols of A and B are similar to those

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

$\mathsf{T}'_{\mathsf{AB}}(z)$
in: $(\mathsf{T}, 1)$
wit: $\perp$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), z\text{₿}), 2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), (1-z)\text{₿})$

Figure 5.5: *Transactions of the escrow contract.*

in Example 5.1.1, except for the part where they ask C for an arbitration:

$$
\begin{aligned}
P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T})\}.\,(\tau.\mathsf{B}\,!\,\mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{B}}) \ + \ \tau.P') \\
P' &= \big(\mathsf{B}\,?\,x.\,(\mathsf{put}\ \mathsf{T}'_{\mathsf{A}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{A}})\,x\} + P''')\big) \ + \ P''' \\
P''' &= \mathsf{C}\,?\,(z,x).\,\big(\mathsf{check}\ z = 1\ .\ \mathsf{put}\ \mathsf{T}'_{\mathsf{A}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{A}})\,x\} \\
&\qquad + \mathsf{check}\ 0 < z < 1\ .\ \big(\mathsf{put}\ \mathsf{T}'_{\mathsf{AB}}(z)\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{AB}}(z))\,x\} + \tau.\mathbf{0}\big) \\
&\qquad + \mathsf{check}\ z = 0\ .\ \mathbf{0}\big)
\end{aligned}
$$

In the summation within P_{A}, participant A internally chooses whether to send her signature to B (so allowing B to redeem 1₿ via T'_{B}), or to proceed with P'. There, A waits to receive either B's signature (which allows A to redeem 1₿ by putting T'_{A} on the blockchain), or a response from the arbiter, in the process P'''. The three cases in the summation of check in P''' correspond, respectively, to the case where A gets a full refund ($z = 1$), a partial refund ($0 < z < 1$), or no refund at all ($z = 0$).

The protocol for B is dual to that of A:

$$
\begin{aligned}
Q_{\mathsf{B}} &= \mathsf{ask}\ \mathsf{T}.\,(\tau.\mathsf{A}\,!\,\mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{A}}) \ + \ \tau.Q') \\
Q' &= \big(\mathsf{A}\,?\,x.\,(\mathsf{put}\ \mathsf{T}'_{\mathsf{B}}\{\mathsf{wit} \mapsto x\,\mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{B}})\} + Q'')\big) \ + \ Q'' \\
Q'' &= \mathsf{C}\,?\,(z,x).\,\big(\mathsf{check}\ z = 0\ .\ \mathsf{put}\ \mathsf{T}'_{\mathsf{B}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{B}})\,x\} \\
&\qquad + \mathsf{check}\ 0 < z < 1\ .\ \big(\mathsf{put}\ \mathsf{T}'_{\mathsf{AB}}(z)\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{AB}}(z))\,x\} + \tau.\mathbf{0}\big) \\
&\qquad + \mathsf{check}\ z = 1\ .\ \mathbf{0}\big)
\end{aligned}
$$

If an arbitration is requested, C internally decides (through the τ actions) who between A and B can redeem the deposit in T, by sending its signature to one of the two participants, or decide for a partial refund of z and $1 - z$ bitcoins, respectively, to A and B, by sending its signature on

T
in: $1 \mapsto (\mathsf{T}_A, 1)$
wit: $\bot$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_1, \varsigma), v_1\text{₿})$ $2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_2, \varsigma), v_2\text{₿})$

T'
in: $1 \mapsto (\mathsf{T}_A, 1), 2 \mapsto (\mathsf{T}_B, 1)$
wit: $1 \mapsto sig_{k_{\mathsf{A}}}^{1*_1}, 2 \mapsto sig_{k_{\mathsf{B}}}^{**}$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_1, \varsigma), v_1\text{₿})$ $2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_2, \varsigma), v_2\text{₿})$

Figure 5.6: *Transactions of the Bitcoin mixing contract.*

$\mathsf{T}'_{\mathsf{AB}}$ to both participants:

$$
\begin{aligned}
R_{\mathsf{C}} &= \mathsf{ask}\ \mathsf{T}.\big(\tau.\mathsf{A}\,!\,(1, \mathsf{sig}_{k_{\mathsf{C}}}^{**}(\mathsf{T}'_{\mathsf{A}})) + \tau.\mathsf{B}\,!\,(1, \mathsf{sig}_{k_{\mathsf{C}}}^{**}(\mathsf{T}'_{\mathsf{B}})) + \tau.R_{\mathsf{AB}}\big) \\
R_{\mathsf{AB}} &= \textstyle\sum_{0<z<1} \tau.\big(\mathsf{A}\,!\,(z, \mathsf{sig}_{k_{\mathsf{C}}}^{**}(\mathsf{T}'_{\mathsf{AB}}(z))) \mid \mathsf{B}\,!\,(z, \mathsf{sig}_{k_{\mathsf{C}}}^{**}(\mathsf{T}'_{\mathsf{AB}}(z)))\big)
\end{aligned}
$$

Note that, in the unlikely case where both A and B choose to send their signature to the other participant, the 1₿ deposit becomes "frozen". In a more concrete version of this contract, a participant could keep listening for the signature, and attempt to redeem the deposit when (unexpectedly) receiving it.

5.2.4 Mixing bitcoins

Assume two participants, A and B (with keys k_{A} and k_{B}, respectively), who want to pay, respectively, v_1₿ to C_1 and v_2₿ to C_2 (with keys k_1 and k_2, respectively). Assume also that the blockchain **B** contains two unspent transactions, T_{A} and T_{B}, such that:

$$
\mathsf{T}_{\mathsf{A}}.\mathsf{out} = (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), v_1\text{₿}) \qquad \mathsf{T}_{\mathsf{B}}.\mathsf{out} = (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), v_2\text{₿})
$$

which can be redeemed by A and B, respectively.

To avoid de-anonymization, A mixes her bitcoins with those of B, using the following protocol:

$$
P_{\mathsf{A}} = \mathsf{B}\,!\,sig_{k_{\mathsf{A}}}^{1*_1}(\mathsf{T})
$$

where the transaction T is displayed in Figure 5.6 (left). The transaction T has one input (the only output of T_{A}), and two outputs: one redeemable with k_1, and the other one with k_2. Participant A sends her signature (computed using the modifier $1*_1$) to participant B. With the modifier $1*_1$, the verification of such signature will consider all outputs, but only the input with index 1. Therefore, B will be able to add inputs to T, without invalidating A's signature.

T_{AB}
in: $(\mathsf{T}_{\mathsf{A}}, v_{\mathsf{C}})$
wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}(k_{\mathsf{A}} k_{\mathsf{B}}, \varsigma\varsigma'), (v_{\mathsf{B}} + v_{\mathsf{C}})\text{₿})$

T_{BC}
in: $(\mathsf{T}_{\mathsf{AB}}, 1)$
wit: $\bot$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), v_{\mathsf{B}}\text{₿})$, $2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{C}}, \varsigma), v_{\mathsf{C}}\text{₿})$

Figure 5.7: *Transactions of the intermediated payment contract.*

Note that T tries to redeem v_1₿ from $\mathsf{T}_{\mathsf{A}}.\mathsf{out}(1)$, but it makes available $v_1 + v_2 > v_1$ bitcoins through its outputs. Hence, T is *not* a consistent update of $\mathbf{B}$, since it violates condition (3) of Definition 3.14. However, since A has signed T with the modifier $1{*}_1$, B can add another input without breaking her signature (but he cannot steal her bitcoins, since all the outputs are signed).

The protocol of B is the following:

$$Q_{\mathsf{B}} \;=\; \mathsf{A}\,?\,x.\,\mathsf{put}\ \mathsf{T}\{\mathsf{in}(2) \mapsto (\mathsf{T}_B, 1)\}\{\mathsf{wit}(1) \mapsto x\}\{\mathsf{wit}(2) \mapsto \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\{\mathsf{in}(2) \mapsto (\mathsf{T}_B, 1)\})\}$$

Here, B completes the transaction T, adding an input, A's signature (in x), and his signature on T (signing all inputs and all outputs, so that the transaction cannot be changed anymore). This modified transaction, displayed in Figure 5.6 (right), is a consistent update of $\mathbf{B}$, hence B can put it on the blockchain.

This contract works as an anonymity schema because it falsifies the premise of the *multi-input* address clustering heuristic, firstly introduced by [61]. It assumes that all the inputs of a transaction are signed using keys controlled by the same user, which is not true in T'.

5.2.5 Intermediated payment

Assume that A wants to send an indirect payment of v_{C}₿ to C, routing it through an intermediary B who retains a fee of $v_{\mathsf{B}} < v_{\mathsf{C}}$ bitcoins. Since A does not trust B, she wants to use a contract to guarantee that: (i) if B is honest, then v_{C}₿ are transferred to C; (ii) if B is *not* honest, then A does not lose money. The contract uses the transactions in Figure 5.7: T_{AB} transfers $(v_{\mathsf{B}} + v_{\mathsf{C}})$₿ from A to B, and T_{BC} splits the amount to B (v_{B}₿) and to C (v_{C}₿). We assume that $(\mathsf{T}_{\mathsf{A}}, 1)$ is a transaction output redeemable by A. The behaviour of A is as follows:

$$\begin{aligned} P_{\mathsf{A}} &= (\mathsf{B}\,?x.\ \mathsf{if}\ \mathsf{versig}_{k_\mathsf{B}}(x, \mathsf{T}_{\mathsf{BC}})\ \mathsf{then}\ P'\ \mathsf{else}\ \mathbf{0}) + \tau \\ P' &= \mathsf{put}\ \mathsf{T}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{AB}})\}.\ \mathsf{put}\ \mathsf{T}_{\mathsf{BC}}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}_{\mathsf{BC}})\, x\} \end{aligned}$$

Here, A receives from B his signature on T_{BC}, which makes it possible to pay C later on. The τ branch and the else branch ensure that A will correctly terminate also if B is dishonest (i.e., B does not send anything, or he sends an invalid signature). If A receives a valid signature, she puts T_{AB} on the blockchain, adding her signature to the wit field. Then, she also appends T_{BC}, adding to the wit field her signature and B's one. Since A takes care of publishing both transactions, the behaviour of B consists just in sending his signature on T_{BC}. Therefore, B's protocol can just be modelled as $Q_{\mathsf{B}} = \mathsf{A}\,!\,\mathsf{sig}^{**}_{k_\mathsf{B}}(\mathsf{T}_{\mathsf{BC}})$.

This contract relies on SegWit. In Bitcoin without SegWit, the identifier of T_{AB} is affected by the instantiation of the wit field. So, when T_{AB} is put on the blockchain, the input in T_{BC} (which was computed before) does not point to it.

5.2.6 Timed commitment

Assume that A wants to choose a secret s, and reveal it after some time — while guaranteeing that the revealed value corresponds to the chosen secret (or paying a penalty otherwise). This can be obtained through a *timed commitment* [29], a protocol with applications e.g. in gambling games [39, 71, 47], where the secret contains the player move, and the delay in the revelation of the secret is intended to prevent other players from altering the outcome of the game. Here we formalise the version of the timed commitment protocol presented in [4].

Intuitively, A starts by exposing the hash of the secret, i.e. $h = H(s)$, and at the same time depositing some amount v₿ in a transaction. The participant B has the guarantee that after t time units, he will either know the secret s, or he will be able to redeem v₿.

The transactions of the protocol are shown in Figure 5.8, where we assume that $(\mathsf{T}_{\mathsf{A}}, 1)$ is a transaction output redeemable by A. The behaviour of A is modelled as the following protocol:

$$\begin{aligned} P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}_{com}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}_{com})\}.\ \mathsf{B}\,!\,\mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}_{pay}).\ P' \\ P' &= \tau\,.\,\mathsf{put}\ \mathsf{T}_{open}\{\mathsf{wit} \mapsto s\ \mathsf{sig}^{**}_{k_\mathsf{A}}(\mathsf{T}_{open})\ \bot\} + \tau \end{aligned}$$

Participant A starts by putting the transaction T_{com} on the blockchain. Note that within this transaction A is committing the hash of the chosen

T_{com}
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\bot$
out: $(\lambda x \varsigma \varsigma'.(\mathsf{versig}(k_{\mathsf{A}}, \varsigma) \text{ and } \mathsf{H}(x) = h)$ or $\mathsf{versig}(k_{\mathsf{A}} k_{\mathsf{B}}, \varsigma\varsigma'), v฿)$

T_{open}
in: $(\mathsf{T}_{com}, 1)$
wit: $\bot$
out: $(\lambda \varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), v฿)$

T_{pay}
in: $(\mathsf{T}_{com}, 1)$
wit: $\bot$
out: $(\lambda \varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), v฿)$
relLock: t

Figure 5.8: *Transactions of the timed commitment.*

secret: indeed, h is encoded within the output script T_{com}.out. Then, A sends to B her signature on T_{pay}. Note that this transaction can be redeemed by B only when t time units have passed since T_{com} has been published on the blockchain, because of the relative timelock declared in T_{pay}.relLock. After sending her signature on T_{pay}, A internally chooses whether to reveal the secret, or do nothing (via the τ actions). In the first case, A must put the transaction T_{open} on the blockchain. Since it redeems T_{com}, she needs to write in T_{open}.wit both the secret s and her signature, so making the former public.

A possible behaviour of the receiver B is the following:

$$Q_{\mathsf{B}} = \big(\mathsf{A}\,?\,x.\ \mathsf{if}\ \mathsf{versig}_{k_{\mathsf{A}}}(x, \mathsf{T}_{pay})\ \mathsf{then}\ Q\ \mathsf{else}\ \mathbf{0}\big) + \tau$$

$$Q = \mathsf{put}\ \mathsf{T}_{pay}\{\mathsf{wit} \mapsto \bot\ x\ \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}_{pay})\} + \mathsf{ask}\ \mathsf{T}_{open}\ \mathsf{as}\ o.\ Q'(get_{secret}(o))$$

In this protocol, B first receives from A (and saves in x) her signature on the transaction T_{pay}. Then, B checks if the signature is valid: if not, he aborts the protocol. Even if the signature is valid, B cannot put T_{pay} on the blockchain and redeem the deposit immediately, since the transaction has a timelock t. Note that B cannot change the timelock: indeed, doing so would invalidate A's signature on T_{pay}. If, after t time units, A has not published T_{open} yet, B can proceed to put T_{pay} on the blockchain, writing A's and his own signatures in the witness. Otherwise, B retrieves T_{open} from the blockchain, from which he can obtain the secret, and use it in Q'.

A variant of this contract, which implements the timeout in T_{com}.out, and does not require the signature exchange, is used in Section 5.2.8.

Timed commitment without signature exchange The timed commitment protocol in Section 5.2.6 has a major drawback: A could put the

T_{com}
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\perp$
out: $(\lambda x\varsigma.(\mathsf{versig}(k_{\mathsf{A}}, \varsigma) \text{ and } \mathsf{H}(x) = h)$ or $\mathsf{relAfter}\ t : \mathsf{versig}(k_{\mathsf{B}}, \varsigma), v\text{₿})$

T_{open}
in: $(\mathsf{T}_{com}, 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), v\text{₿})$

T_{pay}
in: $(\mathsf{T}_{com}, 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), v\text{₿})$
relLock: t

Figure 5.9: *Transactions of the timed commitment without signature exchange.*

transaction T_{com} on the blockchain, and then avoid to send $\mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{pay})$. It this way, even if she started the contract, she could maintain the secret hidden without losing the deposit. We now present a variant of the protocol which addresses this problem. The new transactions are in Figure 5.9.

The new version of T_{com} can be redeemed by B using only his own signature. In order to enforce A to reveal the secret before t time units have passed, the output script of T_{com} uses the expression $\mathsf{relAfter}\ t : \mathsf{versig}(k_{\mathsf{B}}, \varsigma)$. This has the effect of verifying the signature in the witness of the redeeming transaction (i.e., T_{pay}) only if such transaction has relLock greater or equal than t.

The participants' protocols are similar to those in Section 5.2.6, except for the signature exchange, which is no longer necessary.

$$P_{\mathsf{A}} = \mathsf{put}\ \mathsf{T}_{com}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{com})\}.\big(\tau\,.\,(\mathsf{put}\ \mathsf{T}_{open}\{\mathsf{wit} \mapsto s\ \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{open})\}) + \tau\big)$$
$$Q_{\mathsf{B}} = \mathsf{ask}\ \mathsf{T}_{com}.\big(\mathsf{put}\ \mathsf{T}_{pay}\{\mathsf{wit} \mapsto \perp\ \mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}_{pay})\} + \tau\big)$$

5.2.7 Micropayment channels

Assume that A wants to make a series of micropayments to B, e.g. a small fraction of ₿ every few minutes. Doing so with one transaction per payment would result in conspicuous fees [82], so A and B use a micropayment channel contract [108]. A starts by depositing k₿; then, she signs a transaction that pays v₿ to B and $(k - v)$₿ back to herself, and she sends that transaction to B. Participant B can choose to publish that transaction immediately and redeem its payment, or to wait in case A sends another transaction with increased value. A can stop sending

T_{bond}
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}(k_{\mathsf{A}} k_{\mathsf{B}}, \varsigma\varsigma')$ or $\mathsf{relAfter}\ t : \mathsf{versig}(k_{\mathsf{A}}, \varsigma), k\text{₿})$

$\mathsf{T}_{pay}(v)$
in: $(\mathsf{T}_{bond}, 1)$
wit: $\bot$
out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), (k-v)\text{₿})$ $2 \mapsto (\lambda\varsigma.\mathsf{versig}(k_{\mathsf{C}}, \varsigma), v\text{₿})$

T_{ref}
in: $(\mathsf{T}_{bond}, 1)$
wit: $\bot$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), v\text{₿})$
relLock: t

Figure 5.10: *Transactions of the micropayment channel contract.*

signatures at any time. If B redeems, then A can get back the remaining amount. If B does not cooperate, A can redeem all the amount after a timeout.

The protocol of A is the following (the transactions are in Figure 5.10). A publishes the transaction T_{bond}, depositing $k\text{₿}$ that can be spent with her signature and that of B, or with her signature alone, after time t. A can redeem the deposit by publishing the transaction T_{ref}. To pay for the service, A sends to B the amount v she is paying, and her signature on $\mathsf{T}_{pay}(v)$. Then, she can decide to increase v and recur, or to terminate.

$$\begin{aligned}
P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}_{bond}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{bond})\}.\,(P(1) \mid \mathsf{put}\ \mathsf{T}_{ref}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{ref})\})\\
P(v) &= \mathsf{B}\,!\,(v, \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{pay}(v))).\,(\tau + \tau.P(v+1))
\end{aligned}$$

The participant B waits for T_{bond} to appear on the blockchain, then receives the first value v and A's signature σ. Then, B checks if σ is valid, otherwise he aborts the protocol. At this point, B waits for another pair (v', σ'), or, after a timeout, he redeems $v\text{₿}$ using $\mathsf{T}_{pay}(v)$.

$$\begin{aligned}
Q_{\mathsf{B}} &= \mathsf{ask}\ \mathsf{T}_{bond}.\,\mathsf{A}\,?\,(v,\sigma).\,\mathsf{if}\ \mathsf{versig}_{k_{\mathsf{A}}}(\sigma, \mathsf{T}_{pay}(v))\ \mathsf{then}\ P'(v,\sigma)\ \mathsf{else}\ \tau\\
P'(v,\sigma) &= \tau.P_{pay}(v,\sigma)\ +\\
&\qquad \mathsf{A}\,?\,(v',\sigma').\,\mathsf{if}\ v' > v\ \mathsf{and}\ \mathsf{versig}_{k_{\mathsf{A}}}(\sigma', \mathsf{T}_{pay}(v'))\ \mathsf{then}\ P'(v',\sigma')\ \mathsf{else}\ P'(v,\sigma)\\
P_{pay}(v,\sigma) &= \mathsf{put}\ \mathsf{T}_{pay}(v)\{\mathsf{wit} \mapsto \sigma\,\mathsf{sig}^{**}_{k_{\mathsf{B}}}(\mathsf{T}_{pay}(v))\}
\end{aligned}$$

Note that Q_{B} should redeem T_{pay} before the timeout expires, which is not modelled in Q_{B}. This could be obtained by enriching the calculus with time-constraining operators.

5.2.8 Fair lotteries

A multiparty lottery is a protocol where N players put their bets in a pot, and a winner — uniformly chosen among the players — redeems the whole pot. Various contracts for multiparty lotteries on Bitcoin have been proposed in [4, 5, 80, 26, 23, 62]. These contracts enjoy a *fairness* property, which roughly guarantees that: (i) each honest player will have (on average) a non-negative payoff, even in the presence of adversaries; (ii) when all the players are honest, the protocol behaves as an ideal lottery: one player wins the whole pot (with probability $1/N$), while all the others lose their bets (with probability $N-1/N$).

Here we illustrate the lottery in [4], for $N = 2$. Consider two players A and B who want to bet 1₿ each. Their protocol is composed of two phases. The first phase is a timed commitment (as in Section 5.2.6): each player chooses a secret (s_{A} and s_{B}) and commits its hash ($h_{\mathsf{A}} = H(s_{\mathsf{A}})$ and $h_{\mathsf{B}} = H(s_{\mathsf{B}})$). In doing that, both players put a deposit of 2₿ on the ledger, which is used to compensate the other player in case one chooses not to reveal the secret later on. In the second phase, the two bets are put on the ledger. After that, the players reveal their secrets, and redeem their deposits. Then, the secrets are used to compute the winner of the lottery in a fair manner. Finally, the winner redeems the bets.

The transactions needed for this lottery are displayed in Figure 5.11 (we only show A's transactions, as those of B are similar). The transactions for the commitment phase ($\mathsf{T}_{com}, \mathsf{T}_{open}, \mathsf{T}_{pay}$) are similar to those in Section 5.2.6: they only differ in the script of T_{com}.out, which now also checks that the length of the secret is either 128 or 129. This check forces the players to choose their secret so that it has one of these lengths, and reveal it (using T_{open}) before the absLock deadline, since otherwise they will lose their deposits (enabling T_{pay}).

The bets are put using $\mathsf{T}_{lottery}$, whose output script computes the winner using the secrets, which can then be revealed. For this, the secret lengths are compared: if equal, A wins, otherwise B wins. In this way, the lottery is equivalent to a coin toss. Note that, if a malicious player chooses a secret having another length than 128 or 129, the $\mathsf{T}_{lottery}$ transaction will become stuck, but its opponent will be compensated using the deposit.

The endpoint protocol P_{A} of player A follows (the one for B is similar):

$\mathsf{T}_{\mathsf{A}\,com}(h_{\mathsf{A}})$
in: $(\mathsf{T}_{\mathsf{A}\,dep}, 1)$
wit: $\perp$
out: $(\lambda x\varsigma.(\mathsf{versig}(k_{\mathsf{A}},\varsigma) \text{ and } \mathsf{H}(x) = h_{\mathsf{A}} \text{ and } (\lvert x\rvert = 128 \text{ or } \lvert x\rvert = 129)) \text{ or absAfter } t : \mathsf{versig}(k_{\mathsf{B}},\varsigma), 2\text{₿})$

$\mathsf{T}_{lottery}(h_{\mathsf{A}}, h_{\mathsf{B}})$
in: $1 \mapsto (\mathsf{T}_{\mathsf{A}\,bet}, 1), 2 \mapsto (\mathsf{T}_{\mathsf{B}\,bet}, 1)$
wit: $\perp$
out: $(\lambda\varsigma x y.\mathsf{H}(x) = h_{\mathsf{A}} \text{ and } \mathsf{H}(y) = h_{\mathsf{B}} \text{ and } (\lvert x\rvert = 128 \text{ or } \lvert x\rvert = 129) \text{ and } (\lvert y\rvert = 128 \text{ or } \lvert y\rvert = 129) \text{ and if } \lvert x\rvert = \lvert y\rvert \text{ then } \mathsf{versig}(k_{\mathsf{A}},\varsigma) \text{ else } \mathsf{versig}(k_{\mathsf{B}},\varsigma), 2\text{₿})$

$\mathsf{T}_{\mathsf{A}\,open}(h_{\mathsf{A}})$
in: $(\mathsf{T}_{\mathsf{A}\,com}(h_{\mathsf{A}}), 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}},\varsigma), 2\text{₿})$

$\mathsf{T}_{\mathsf{A}\,pay}(h_{\mathsf{A}})$
in: $(\mathsf{T}_{\mathsf{A}\,com}(h_{\mathsf{A}}), 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}},\varsigma), 2\text{₿})$
absLock: t

$\mathsf{T}_{\mathsf{A}\,win}(h_{\mathsf{A}}, h_{\mathsf{B}})$
in: $(\mathsf{T}_{lottery}(h_{\mathsf{A}}, h_{\mathsf{B}}), 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}},\varsigma), 2\text{₿})$

Figure 5.11: *Transactions of the fair lottery with deposit.*

$$
\begin{aligned}
P_{\mathsf{A}} &= \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,com}\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{A}\,com})\}.\big(\mathsf{ask}\ \mathsf{T}_{\mathsf{B}\,com}\ \mathsf{as}\ y.\,P' \;+\; \tau.P_{open}\big)\\
P' &= \mathsf{let}\ h_{\mathsf{B}} = get_{hash}(y)\ \mathsf{in\ if}\ h_{\mathsf{B}} \neq h_{\mathsf{A}}\ \mathsf{then}\ P_{pay} \mid P'''\ \mathsf{else}\ P_{pay} \mid P_{open}\\
P''' &= \mathsf{B}\,?\,x.\,P'' \;+\; \tau.P_{open}\\
P'' &= \mathsf{let}\ \sigma = \mathsf{sig}^{**,1}_{k_{\mathsf{A}}}(\mathsf{T}_{lottery}(h_{\mathsf{A}}, h_{\mathsf{B}}))\ \mathsf{in}\\
&\quad \big(\mathsf{put}\ \mathsf{T}_{lottery}(h_{\mathsf{A}}, h_{\mathsf{B}})\{\mathsf{wit}(1) \mapsto \sigma\}\{\mathsf{wit}(2) \mapsto x\}.\,(P_{open} \mid P_{win})\big) + \tau.P_{open}\\
P_{pay} &= \mathsf{put}\ \mathsf{T}_{\mathsf{B}\,pay}\{\mathsf{wit} \mapsto \perp\ \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{B}\,pay})\}\\
P_{open} &= \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,open}\{\mathsf{wit} \mapsto s_{\mathsf{A}}\ \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{A}\,open})\}\\
P_{win} &= \mathsf{ask}\ \mathsf{T}_{\mathsf{B}\,open}\ \mathsf{as}\ z.\,P'_{win}\\
P'_{win} &= \mathsf{put}\ \mathsf{T}_{\mathsf{A}\,win}(h_{\mathsf{A}}, h_{\mathsf{B}})\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{\mathsf{A}\,win}(h_{\mathsf{A}}, h_{\mathsf{B}}))\ s_{\mathsf{A}}\ get_{secret}(z)\}
\end{aligned}
$$

Player A starts by putting $\mathsf{T}_{\mathsf{A}\,com}$ on the blockchain, then she waits for B doing the same. If B does not cooperate, A can safely abort the protocol taking its $\tau.P_{open}$ branch, so redeeming her deposit with $\mathsf{T}_{\mathsf{A}\,open}$ (as usual, here with τ we are modelling a timeout). If B commits his secret, A executes P', extracting the hash h_{B} of B's secret, and checking whether it is distinct from h_{A}. If the hashes are found to be equal, A aborts the protocol using P_{open}. Otherwise, A runs $P''' \mid P_{pay}$. The P_{pay} component attempts to redeem B's deposit, as soon as the absLock deadline of $\mathsf{T}_{\mathsf{B}pay}$ expires, forcing B to timely reveal his secret. Instead, P''' proceeds with the lottery, asking B for his signature of $\mathsf{T}_{lottery}$. If B does not sign, A aborts using P_{open}. Then, A runs P'', finally putting the bets ($\mathsf{T}_{lottery}$) on the ledger. If this is not possible (e.g., because one of the T_{bet} is already spent), A aborts using P_{open}. After $\mathsf{T}_{lottery}$ is on the ledger, A reveals her secret and redeems her deposit with P_{open}. In parallel, with P_{win} she waits for the secret of B to be revealed, and then attempts to redeem the pot ($\mathsf{T}_{\mathsf{A}win}$).

The fairness of this lottery has been established in [4]. This protocol can be generalised to $N > 2$ players [4, 5] but in this case the deposit grows quadratically with N. The works [62, 23] have proposed fair multiparty lotteries that require, respectively, zero and constant (≥ 0) deposit. More precisely, [62] devises two variants of the protocol: the first one only relies on SegWit, but requires each player to statically sign $O(2^N)$ transactions; the second variant reduces the number of signatures to $O(N^2)$, at the cost of introducing a custom opcode. Also the protocol in [23] assumes an extension of Bitcoin, i.e. the malleability of in fields, to obtain an ideal fair lottery with $O(N)$ signatures per player (see Section 7.3).

5.2.9 Contingent payments

Assume a participant A who wants to pay v₿ to receive a value s which makes a public predicate p true, where $p(s)$ can be verified efficiently. A seller B who knows such s is willing to reveal it to A, but only under the guarantee that he will be paid v₿. Similarly, the buyer wants to pay only if guaranteed to obtain s.

A naïve attempt to implement this contract in Bitcoin is the following: A creates a transaction T such that $\mathsf{T}.\mathsf{out}(\varsigma, x)$ evaluates to true if and only if $p(x)$ holds and ς is a signature of B. Hence, B can redeem v₿ from T by revealing s. In practice, though, this approach is arguably useful, since it requires coding p in the Bitcoin scripting language, whose expressiveness is quite limited.

More general contingent payment contracts can be obtained by exploiting zero-knowledge proofs [12, 115, 38]. In this setting, the seller generates a fresh key k, and sends to the buyer the encryption $e_s = E_k(s)$, together with the hash $h_k = H(k)$, and a zero-knowledge proof guaranteeing that such messages have the intended form. After verifying this proof, A is sure that B knows a preimage k' of h_k (by collision resistance, $k' = k$) such

$\mathsf{T}_{cp}(h)$
in: $(\mathsf{T}_{\mathsf{A}}, 1)$
wit: $\perp$
out: $(\lambda x\varsigma.(\mathsf{versig}(k_{\mathsf{B}}, \varsigma)\ \mathsf{and}\ \mathsf{H}(x) = h)$ $\mathsf{or\ relAfter}\ t : \mathsf{versig}(k_{\mathsf{A}}, \varsigma), v$₿$)$

$\mathsf{T}_{open}(h)$
in: $(\mathsf{T}_{cp}(h), 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{B}}, \varsigma), v$₿$)$

$\mathsf{T}_{refund}(h)$
in: $(\mathsf{T}_{cp}(h), 1)$
wit: $\perp$
out: $(\lambda\varsigma.\mathsf{versig}(k_{\mathsf{A}}, \varsigma), v$₿$)$
relLock: t

Figure 5.12: *Transactions of the contingent payment.*

that $D_{k'}(e_s)$ satisfies the predicate p, and so she can buy the preimage k of h_k with the naïve protocol, so obtaining the solution s by decrypting e_s with k.

The transactions implementing this contract are displayed in Figure 5.12. The relAfter : clause in T_{cp} allows A to redeem v₿ if no solution is provided by the deadline t. The behaviour of the buyer A can be modelled as follows:

$$
\begin{aligned}
P_{\mathsf{A}} &= \mathsf{B}\,?\,(e_s, h_k, z).\,P \ + \ \tau \\
P &= \mathsf{if}\ \mathit{verify}(e_s, h_k, z)\ \mathsf{then\ put}\ \mathsf{T}_{cp}(h_k)\{\mathsf{wit} \mapsto \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{cp}(h_k))\}.\,P'\ \mathsf{else}\ \mathbf{0} \\
P' &= \mathsf{ask}\ \mathsf{T}_{open}(h_k)\ \mathsf{as}\ x.\,P'''(D_{get_k(x)}(e_s)) \ + \\
&\qquad \mathsf{put}\ \mathsf{T}_{refund}(h_k)\{\mathsf{wit} \mapsto \perp\ \mathsf{sig}^{**}_{k_{\mathsf{A}}}(\mathsf{T}_{refund}(h_k))\})
\end{aligned}
$$

For simplicity, here we model the zero-knowledge proof as a single message. More concretely, it should be modelled as a sub-protocol. Upon receiving e_s, h_k and the proof $z$the buyer verifies z. If the verification succeeds, A puts $\mathsf{T}_{cp}(h_k)$ on the blockchain. Then, she waits for T_{open}, from which she can retrieve the key k, and so use the solution $D_{get_k(x)}(e_s)$ in P'''. In this way, B can redeem v₿. If B does not put T_{open}, after t time units A can get her deposit back through T_{refund}. The protocol of B is simple, so it is omitted.

Chapter 6

A Toolchain for Developing BitML Contracts

In the last five years much outstanding research has been devoted to showing how to exploit Bitcoin to execute *smart contracts* — computer protocols which allow for exchanging cryptocurrency according to complex pre-agreed rules [4, 3, 1, 5, 12, 23, 26, 56, 57, 55, 58, 62]. Despite the wide variety of use cases witnessed by these works, no tool support has been provided yet to facilitate the development of multi-transaction Bitcoin contracts. Today, this task requires to devise complex protocols which, besides using the standard cryptographic primitives, (e.g., encryption, hash functions, signatures), can read and append transactions on the Bitcoin blockchain. Creating a new protocol requires a significant effort to establish its correctness and security: this is an error-prone task, usually performed manually, with the risk of overlooking some corner cases. Crafting the transactions used by these protocols is burdensome as well, since it requires to struggle with low-level, poorly documented features of Bitcoin, like e.g. its stack-based scripting language.

We consider BitML, which features a computationally sound embedding into Bitcoin [22], and a sound and complete verification technique of relevant trace properties [24]. BitML can express many of the smart contracts appeared in the literature [15, 9], and execute them by appending suitable transactions to the Bitcoin blockchain. The computational soundness of the embedding guarantees that security properties at the level of the BitML semantics are preserved at the level of Bitcoin transactions, even in the presence of adversaries. Still, BitML lives in a theoretical limbo, as no tool support exists yet to develop contracts and deploy them on the Bitcoin blockchain.

We develop a toolchain for writing and verifying BitML contracts, and for deploying them on Bitcoin. More specifically, the toolchain is composed of:

1. A BitML embedding in Racket [42], which allows for programming BitML contracts within the DrRacket IDE.

2. A security analyser which can check arbitrary LTL properties of BitML contracts. In particular, the analysis can decide *liquidity*, a landmark property of smart contracts requiring that the funds within a contract do not remain frozen forever.

3. A compiler from BitML contracts to standard Bitcoin transactions. The computational soundness result in [22] ensures that attacks to compiled contracts are also observable at the BitML level. Therefore, the properties verified by our security analyzer also hold for compiled contracts.

The architecture of our toolchain is displayed in Figure 6.1. The development workflow is the following: (a) write the BitML contract, and specify the required properties. Optionally, specify some constraints on the participants' strategies, e.g. to partially define the behaviour of the honest participants; (b) verify that the contract satisfies the required properties through the security analyser; (c) compile the contract to Bitcoin transactions; (d) execute the contract, by appending these transactions to the Bitcoin blockchain according to the chosen strategy. The first two steps are performed within the DrRacket IDE, while the third one requires participants to use the Balzac tool to compute signatures and compile Balzac transactions into Bitcoin transactions. We remark that the last step can be performed on the Bitcoin main network, without requiring any extensions or customizations.

The toolchain also supports an extension of BitML that enables recursion and renegotiation of contracts [16]. In this variant, the security analyser implements a different semantics, and can only check the liquidity of a contract.

We provide a collection of BitML contracts [93], which we use as a benchmark to evaluate our toolchain. This collection contains some of the most complex contracts ever developed for Bitcoin, e.g. financial services, auctions, timed commitments, lotteries, and a variety of other gambling games. We use our benchmarks to discuss the expressiveness and the limitations of Bitcoin as a smart contracts platform.

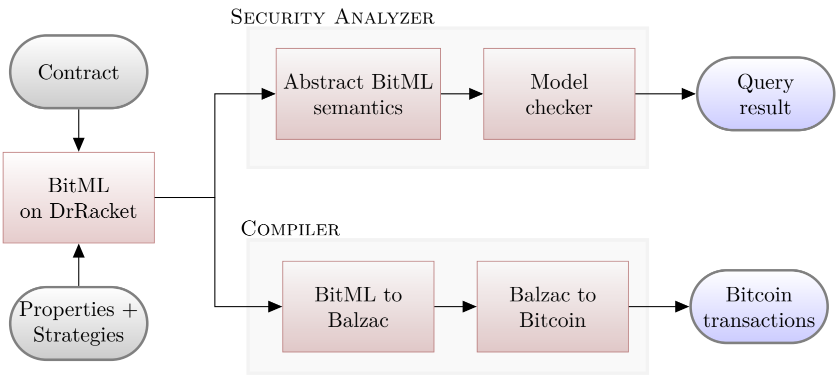


Figure 6.1: *Toolchain architecture.*

All the components of our toolchain are open-source [92, 96, 81], as well as the contracts in our benchmark. This Chapter is accompanied by a brief video that introduces the features of the toolchain (*Demo Video URL:* `https://bit.ly/2MxOBsT`).

6.1 Designing BitML contracts

BitML contracts allow two or more participants to exchange their bitcoins (₿) according to a given logic. A contract consists of two parts: a precondition, describing requirements that participants must fulfil to stipulate the contract, and a process, which specifies the execution logic of the contract. Here, rather than providing the syntax and semantics of BitML (see [22] for a formalization), we illustrate it through a simple but paradigmatic example, the *mutual timed commitment* contract [4]. This contract involves two participants (named below `A` and `B`) each one choosing a secret and depositing a certain amount of cryptocurrency (say, 1₿). The goal of the contract is to ensure that each participant will either learn the other participant's secret, or otherwise receive the other participant's deposit as a compensation. The contract gives some time to the participants to reveal their secrets. If a participant reveals her secret in time, then she can get her deposit back; otherwise, after the time is up, the other participant can withdraw that deposit.

In our tool, we can specify this contract as follows:

```
(participant "A" "029c...cced") ; A's public key
(participant "B" "022c...af30") ; B's public key

(contract
  (pre
    (deposit "A" 1 "1a34...6f38@0") ; tx output id (1BTC)
    (secret  "A" a "628f...de71")   ; hash of A's secret
    (deposit "B" 1 "19e7...85ff@2") ; tx output id (1BTC)
    (secret  "B" b "9d48...bb35"))  ; hash of B's secret

  (choice
    (reveal (a) (choice
                  (reveal (b) (split
                                (1 -> (withdraw "A"))
                                (1 -> (withdraw "B"))))
                  (after 100050 (withdraw "A"))))
    (after 100000 (withdraw "B")))
```

The first two lines create aliases for the participant names, specifying their public keys. The contract preconditions are in the `pre` part: each participant must specify the identifier of a transaction output, and the hash of the chosen secret. The transaction output must be unspent, must contain the required 1₿, and must be redeemable using the participant's private key. The hash is used during the contract execution: when the participant provides a value, claiming that it is the chosen secret, the hash of this value is required to be equal to the one in the precondition.

The contract logic is specified after the preconditions. The top-level `choice` defines two alternative branches of the contract. The first branch can only be taken if `A` reveals her secret (named `a`); when this happens, the contract continues with the innermost `choice`. The second branch can only be taken after a timeout, specified as the block at height 100000, and it allows `B` to redeem all the funds deposited within the contract (i.e., 2₿) by executing `withdraw "B"`. So, to avoid losing her deposit, `A` is incentivized to reveal her secret in time. Similarly, the innermost `choice` is used to incentivize `B` to reveal his secret before the block at height 100050. If `B` reveals, then the `split` subcontract is executed: this divides the balance of the contract in two parts of 1₿ each, allowing the participants to withdraw their deposits back.

The language is defined exploiting the Racket macro system, which is used to rewrite BitML syntactic constructs to Racket code. This approach benefits from the Racket language ecosystem, and allows us to write BitML contracts in the DrRacket IDE. Indeed, our toolchain integrates within the DrRacket IDE the contract editor, the security analyzer and the BitML compiler. The implementation of BitML in Racket extends the idealized version of BitML in [22] to make the language usable in practice. For instance, it introduces special deposits of type `fee`, which are automatically spread over all the transactions obtained by the compiler. We also implement static checks for a number of errors that could prevent the correct execution of contracts, e.g. committing secrets with the same hash, double spending a transaction output, etc.

6.2 Verifying BitML contracts

The tool verifies various forms of *liquidity*, requiring that no funds (or funds up-to a certain amount) are frozen forever within a contract. Further, the tool can verify arbitrary LTL formulae, where state predicates can specify, e.g., the funds owned by participants, the provided authorizations, and the revealed secrets. By default, the tool verifies the required property against *all* possible behaviours of each participant: for instance, if a contract contains `reveal a`, the verifier considers both the case where the secret is revealed and the one where it is not. Authorizations are handled similarly, by considering both cases. However, in most cases, a participant wishes to verify a contract with respect to a given behaviour for herself, making no assumptions on the other participants' behaviour (unless some other participants are considered trusted, in which case it would make sense to fix a behaviour also for them). For instance, a participant `A` may want to give her authorization to perform a given branch only after participant `B` has revealed his secret. The tool allows for constraining the behaviour of participants, specifying the conditions upon which secrets are revealed and authorizations are provided. Actions which can be performed by everyone, like `withdraw` and `split`, cannot be constrained.

For instance, we can verify that the mutual timed commitment contract is liquid whatever strategies are chosen by participants. The query `check-liquid` correctly answers true, since:

- if `A` does not reveal, then anyone (after the block at height 100000) can perform `withdraw "B"`, which transfers the whole contract balance to `B`;
- if `A` reveals but `B` does not reveal, then anyone (after the block at height 100050) can perform `withdraw "A"`, which transfers the whole contract balance to `A`;
- if both `A` and `B` reveal, then anyone can perform `split`, which transfers the balance in equal parts to `A` and `B`.

Note that if we remove the `after` branch at line 16, the contract is no longer liquid. However, it becomes liquid when `A`'s strategy is to reveal the secret. We can verify that this holds through the query `check-liquid (strategy "A" (do-reveal a))`. Liquidity is lost again if `A` chooses to reveal only after `B` has revealed, i.e. when her strategy is `"A" (do-reveal a) if ("B" (do-reveal b))`.

Besides liquidity, we can check specific LTL properties of contracts through the command `check-query`. E.g., in the mutual timed commitment we can verify that, after `A` reveals, she will eventually get back at least her 1₿ deposit. In LTL, this property is formalised as the following formula, where 10^8 satoshi = 1₿:

```
[](a revealed => <>A has-deposit>= 100000000 satoshi)
```

We also verify that if `A` reveals the secret, then eventually either `B` reveals, or `A` will get `B`'s deposit, too. The LTL query is the following:

```
[](a revealed =>
<>(b revealed \/ A has-deposit>= 200000000 satoshi))
```

Our verification technique is based on model-checking the state space of BitML contracts. Since this state space is infinite, before running the model-checker we reduce it to a finite-state one, by exploiting the abstraction in [24]. This abstraction resolves the three sources of infiniteness of the concrete semantics of BitML: the passing of time, the advertisement/stipulation of contracts, and the off-contract bitcoin transfers. To obtain a finite-state system, the abstraction: (i) quotients time in a finite number of time intervals, (ii) disables the advertisement of new contracts, and (iii) limits the off-contract operations to those for transferring funds to contracts and for destroying them. This abstraction is shown in [24] to enjoy a strict correspondence with the concrete BitML semantics: namely, each concrete step of the contract under analysis is mimicked by an abstract step, and vice versa.

Our tool implements the abstract BitML semantics in Maude, a model-checking framework based on rewriting logic [36]. Maude is particularly convenient for this purpose: we use its equational logic to express structural equivalence between BitML terms, and its conditional rewriting rules to encode the abstract semantics of BitML. In this way, we naturally obtain an executable abstract semantics of BitML. Once a BitML contract in translated in Maude, we use the Maude LTL model-checker [40] to verify the required security properties, under the strategies specified by the user. The various forms of liquidity are also translated to corresponding LTL formulae. The computational soundness of the BitML compiler guarantees that the properties verified by the model checker are preserved when executing the contract on Bitcoin.

6.3 Compiling BitML to Bitcoin

Our compiler operates in two phases: first, it translates BitML contracts into Balzac [81], an abstraction layer over Bitcoin transactions based on the formal model of [11]; then, it translates Balzac transactions into standard Bitcoin transactions.

The compiler from BitML to Balzac implements the algorithm in [22], extending it with transaction fees. In particular, the compiler guarantees that each transaction contains enough fees to be publishable in the blockchain.

The compiler from Balzac to Bitcoin produces *standard* Bitcoin transactions [85]: this is crucial since non-standard ones are discarded by the Bitcoin network. To this purpose, Balzac produces standard output scripts of the form "Pay to Public Key Hash" (P2PKH) or "Pay to Script Hash" (P2SH). P2PKH is used for encoding signature verification (e.g., to redeem the deposit obtained by a `withdraw`), while P2SH is used for complex redeeming conditions (e.g., to check that the revealed secret matches the committed hash). Since Bitcoin requires that all the values pushed by standard scripts fit within 520 bytes, our compiler checks that this constraint is satisfied for each generated script. Balzac outputs serialized raw transactions, which can be directly broadcast to the Bitcoin network.

6.4 Evaluation

To evaluate our toolchain, we use a benchmark of representative use cases, including financial contracts [68, 28], auctions, lotteries [5, 62] and gambling games [91]. For each contract in the benchmark, we display in Table 6.1 the number N of involved participants, the number T of transactions obtained by the compiler, and the verification time V for checking liquidity. For uniformity, in the performance evaluation we focus on liquidity (other queries to verify the functional correctness of the contracts in Table 6.1 are on the repository). We carry out our experiments on a PC equipped with a hexa-core Intel Core i7-7800X CPU @ 3.50GHz, and 64GB of RAM. The participants' strategies are constrained only as needed to ensure liquidity: in most cases, we do not put any constraints at all. For the contracts which involve predicates on secrets (e.g., all the lotteries), in principle one would need to check liquidity against all the possible choices of secrets. To make verification feasible, since each contract only checks a finite set of predicates, we partition the infinite choices of secrets into a finite set of regions, and sample one choice from each region. In this way,

Contract	$\boldsymbol{N}$	$\boldsymbol{T}$	$\boldsymbol{V}$
Mutual timed commitment	2	15	83ms
Mutual timed commitment	3	34	103ms
Mutual timed commitment	4	75	454ms
Mutual timed commitment	5	164	13s
Escrow (early fees)	3	12	8s
Escrow (late fees)	3	11	3.4s
Zero Coupon Bond	3	8	86ms
Coupon Bond	3	18	1.3s
Future(C)	3	$5 + T_C$	80ms $+ V_C$
Option(C, D)	3	$14 + T_C + T_D$	90ms $+ V_C + V_D$
Lottery ($O(N^2)$ collateral)	2	15	427ms
Lottery (0 collateral)	2	8	142ms
Lottery (0 collateral)	4	587	67h
Rock-Paper-Scissors	2	23	781ms
Morra game	2	40	674ms
Shell game	2	23	27s
Auction (2 turns)	2	42	3.3s

Table 6.1: *Benchmarks for the BitML toolchain.*

the liquidity check is performed a finite number of times, ensuring that the verifier explores every reachable state of the contract. For instance, in the 4-players lottery we explore 3^4 regions, which explains the 67 hours needed to verify its liquidity. Another feature which significantly affects the verification time is the fact that we are considering *all* the possible strategies of *all* the participants.

We also evaluate the same set of benchmark contract with the variant of the analyser that supports recursion. The verification time for all the benchmarks is in the order of milliseconds on a consumer-grade laptop. This is due to the fact that the semantics is more abstract (e.g. it does not consider secrets values) and does not support LTL queries or other variant of liquidity other than the plain one.

The only work against which we can compare the performance of our tool is [3], which models Bitcoin contracts in Uppaal, a model-checking framework based on Timed Automata. The most complex contract modelled in [3] is the mutual timed commitment with 2 participants: this requires $\sim$ 30s to be verified in Uppaal, while our tool verifies the same property in $<$ 100ms. This speedup is due to the higher abstraction level of BitML over [3], which operates at the (lower) level of Bitcoin transactions.

One of the main difficulties that we have encountered in developing contracts is that some complex BitML specifications can not be compiled to Bitcoin, because Bitcoin has a 520-byte limit on the size of each value pushed to the evaluation stack [76]. In some cases, we managed to massage the BitML contract so to make its compilation respect the 520-byte constraint. For instance, a common pattern that easily violates the 520-byte constraint is the following:

```
(choice (revealif (b) (pred (p0)) (C0))
        (revealif (b) (pred (p1)) (C1))
        (after T      (C2)))
```

The `choice` is compiled into a transaction whose redeem script encodes the disjunction of *three* logical conditions, corresponding to the three branches of the `choice`. Depending on the predicates `p0` and `p1`, and on the number of participants in the contract, this script may violate the 520-byte constraint. A workaround is to rewrite the pattern above into the following one:

```
(choice (revealif (b) (pred (p0)) (C0))
        (after T (tau (choice
                      (revealif (b) (pred (p1)) (C1))
                      (after T1     (C2))))))
```

In this case the compilation includes two transactions, corresponding to the two `choice`s. The scripts of these transactions encode the disjunction of *two* logical conditions, corresponding to the two branches of the `choice`s. Using this workaround we have managed to compile the 4-players lottery into standard transactions, at the price of increasing the number of transactions (587 for the standard version *vs.* 138 for the nonstandard one). Similar techniques (e.g. simplification of predicates) allowed us to compile all the contracts in Table 6.1 into standard Bitcoin transactions.

In general, the 520-byte constraint intrinsically limits the expressiveness of Bitcoin contracts: for instance, since public keys are 33 bytes long, a contract which needs to simultaneously verify 15 signatures can not be implemented using standard transactions.

6.5 BitML toolchain tutorial

The following Section describes how to exploit the toolchain to design, compile and verify BitML contracts. We proceed in an incremental fashion: each step introduces a new feature through a small, self-enclosed use-case. Some lines have been shortened due to space constraints, refer to [94] for the full examples.

6.5.1 Developing BitML contracts in a nutshell

BitML contracts allow two or more participants to exchange their bitcoins according to complex pre-agreed rules.

The first step in designing a BitML contract is to declare the involved participants. For instance, we can declare three participants `"A"`, `"B"` and `"C"` as follows:

```
(participant "A" "029c5f6...095f547799a6289fbc90c70209c1cced")
(participant "B" "0316589...a876e4f7315fa20a07114d5fb8866553")
(participant "C" "03c7e15...928d986b26fe0dc2533f304c19268a2f")

(debug-mode)
```

Each participant is associated to a public key: for instance, `"A"` has the public key `"029c...cced"`. The command `(debug-mode)` is needed to generate auxiliary keys which are used by the BitML compiler, instead of declaring them as you are supposed to when executing a contract in a real life scenario.

Simple payments Assume that `"A"` simply wants to donate 1₿ to `"B"`. To this purpose, `"A"` must first declare that she owns a transaction output with 1₿. We can define this transaction output as follows:

```
(define (txA) "tx:0200000000...c6bdb51600@1")
```

In the definition above, `"0200000000...c6bdb51600"` are the bytes of the serialized transaction, and the trailing `"@1"` is the index of the output.

The contract advertised by `"A"` is the following:

```
(contract
  (pre (deposit "A" 1 (ref (txA))))
  (withdraw "B"))
```

The contract precondition `(pre (deposit "A"1 (ref (txA))))` declares that `"A"` agrees to transfer the 1₿ referenced by the transaction output `txA` under the control of the contract. The actual contract is `( withdraw "B")`: this just transfers the funds deposited into the contract to `"B"`.

In the previous contract, the initial deposit has been provided by a transaction output; more in general, a contract can gather money from more than one transaction. For instance, assume that another participant `"C"` wants to contribute 1₿ to the donation. The contract precondition is modified as follows:

```
(contract
  (pre (deposit "A" 1 (ref (txA)))
       (deposit "C" 1 "tx:0200000...3b6cdef8ac00000000@0"))
  (withdraw "B"))
```

Procrastinating payments Assume now that `"A"` wants to donate 1₿ to `"B"`, but only after a certain time `t`. For instance, the 1₿ could be a birthday present to be withdrawn only after the birthday date; or the amount of a rent to the landlord, to be paid only after the 1st of the month. We represent the time `t` as a block height. For instance, we set `t` to 500000 (note that the block at this height was actually mined on 2017-12-18).

To craft this contract we use the primitive `after height contract`, which locks the `contract` until the block at the given `height` is appended to the blockchain. We also reuse the transaction output `txA` from the previous example:

```
(define (t) 500000)

(contract
  (pre (deposit "A" 1 (ref (txA))))
  (after (ref (t)) (withdraw "B")))
```

This contract ensures that only after the block at height `t` has been appended to the blockchain, `"B"` will be able to redeem 1₿ from the contract, by performing the action `(withdraw "B")`.

The following contract allows `"A"` to recover her deposit if `"B"` has not withdrawn within a given deadline `t1` $>$ `t`:

```
(define (t) 500000)
(define (t1) 510000)

(contract
  (pre (deposit "A" 1 (ref (txA))))

  (choice
    (after (ref (t)) (withdraw "B"))
    (after (ref (t1)) (withdraw "A"))))
```

The contract allows two (mutually exclusive) behaviours: either `"A"` or `"B"` can withdraw 1₿. Before the deadline `t` no one can withdraw; after `t` (but before `t1`) only `"B"` can withdraw, while after the `t1` both withdraw actions are enabled, so the first one who performs their withdraw will get the money.

Authorizing payments Assume that `"A"` is willing to pay 1₿ to `"A"`, but only if an `"Oracle"` gives his authorization. We can use the authorization primitive `auth Participant Contract` as follows:

```
(contract
  (pre (deposit "A" 1 (ref (txA))))
  (auth "Oracle" (withdraw "B")))
```

This contract ensures that `(withdraw "B")` is performed whenever `"Oracle"` authorizes it.

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

```
(define (Naive-escrow)
  (choice
    (auth "A" (withdraw "B"))
    (auth "B" (withdraw "A"))))
```

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

```
(define (Oracle-escrow)
  (choice
    (auth "A" (withdraw "B")) ; same as
    (auth "B" (withdraw "A")) ; Naive-escrow
    (auth "O" (withdraw "A"))
    (auth "O" (withdraw "B"))))

(contract
  (pre (deposit "A" 1 (ref (txA))))
  (ref (Oracle-escrow)))
```

The last two branches are used if neither "A" nor "B" give their authorizations: in this case, the arbiter chooses whether to authorize "A" or "B" to redeem the deposit.

Splitting deposits In all the previous examples, the deposit within the contract is transferred to a single participant. More in general, deposits can be split in many parts, to be transferred to different participants. For instance, assume that "A" wants her 1₿ deposit to be transferred in equal parts to "B1" and to "B2". We can model this behaviour as follows:

```
(define (Pay-split)
  (split
    (0.5 -> (withdraw "B1"))
    (0.5 -> (withdraw "B2"))))
```

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

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

We can model this behaviour as follows:

```
(contract
  (pre (deposit "A" 1 (ref (txA))))
  (choice
    (auth "I" (split (0.1 -> (withdraw "I"))
                     (0.9 -> (withdraw "B"))))
    (after (ref (d)) (withdraw "A"))))
```

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

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

```
(pre (deposit "A" 1 (ref (txA1)))
     (vol-deposit "A" x 1 (ref (txA2))))
```

gives "A" the possibility of contributing 1Ƀ during the contract execution. However, "A" can choose instead to spend her volatile deposit outside the contract. The variable x is a handle to the volatile deposit, which can be used as follows:

```
(define (Pay?)
  (put (x) (withdraw "B")))
```

Since x is not paid upfront, there is no guarantee that x will be available when the contract demands it, as "A" can spend it for other purposes.

Volatile deposits can be exploited within more complex contracts, to handle situations where a participant wants to add some funds to the contract. For instance, assume a scenario where "A1" and "A2" want to give "B" 2Ƀ as a present, paying 1Ƀ each. However, "A2" is not sure a priori she will be able to pay, because she may need her 1Ƀ for more urgent purposes: in this case, "A1" is willing to pay an extra bitcoin. We can model this scenario as follows: "A1" puts 2Ƀ as a persistent deposit, while "A2" makes available a volatile deposit x of 1Ƀ:

```
(contract
  (pre (deposit "A1" 2 (ref (txA1)))
       (vol-deposit "A2" x 1 (ref (txA2))))
  (choice
    (put (x) (split (2 -> (withdraw "B"))
                    (1 -> (withdraw "A1"))))
    (after 700000 (withdraw "B"))))
```

In the first branch, "A2" puts 1Ƀ in the contract, and the balance is split between "B" (who takes 2Ƀ, as expected), and "A1" (who takes her extra deposit back). The second branch is enabled after d, and it deals with the case where "A2" has not put her deposit by such deadline. In this case, "B" can redeem 2Ƀ, while "A2" loses the extra deposit. Note that, in both cases, "B" will receive 2Ƀ.

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

```
(pre (secret "A" a "f9292914bfd27c426a23465fc122322abbdb63b7")
```

where `"A"` is the participant who owns the secret, `a` is its *name*, and `"f9292914bfd27c426a23465fc122322abbdb63b7"` is its `hash160` hash. We never denote the value of the secret itself. A basic contract which exploits this feature is the following:

```
(define (PaySecret)
  (revealif (a) (pred (> a 1)) (withdraw "A")))
```

This contract asks `"A"` to commit to a secret of length greater than one, as stated in the predicate `(pred (> a 1))`. After revealing `a`, it allows `"A"` to redeem 1 ₿ upon revealing the secret. Until then, the deposit is frozen.

To reveal a secret without imposing a predicate use `(reveal)`. e.g.: `(reveal (a)(withdraw "A"))`

Note that we never refer to the value itself of the secret, rather we use its length. After compiling to Bitcoin, the actual length of the secret will be increased by η, where η is a security parameter, large enough to avoid brute-force preimage attack.

6.5.2 Verifying contracts with the BitML toolchain

Other than compiling contracts to transactions, the BitML toolchain allows to verify them before their execution.

A desirable property of smart contracts is *liquidity*, which requires that the contract balance is always eventually transferred to some participant. In a non-liquid contract, funds can be frozen forever, unavailable to anyone, hence effectively destroyed. There are many possible flavours of liquidity, depending e.g. on which participants are assumed to be honest, and on which are their strategies.

The toolchain can also verify arbitrary security proprieties, expressed as LTL queries.

Liquidity In the following contract, "A" and "B" contribute 1 ₿ each for a donation of 2 ₿ to either "C" or "D". We want to check if the contract is liquid or not, without supplying any strategy, i.e. without knowing which branch "A" and "B" will authorize.

This flavour of liquidity is called *strategy-less.* Intuitively, it corresponds to check if the contract is liquid for any possible strategy of any participants, whether they are honest or not.

To check the liquidity of the following contract, we add (check-liquid) at its end.

```
(participant "A" "0339bd7fade9167e09681...2bde1d57247fbe1")
(participant "B" "034a7192e922118173906...da4f85701a5f809")
(participant "C" "034f5ca30056b9dd89132...fea9c3467631e6b")
(participant "D" "037b60c121050e1fa6e7d...da1c63f0e2a157e")

(debug-mode)

(contract
  (pre
    (deposit "A" 1 "txid:2e647d8566f00a08d27...b2965e35@1")
    (deposit "A" 1 "txid:625bc69c467b33e2abj...bc710168@0"))

  (choice
    (auth "A" "B" (withdraw "C"))
    (auth "A" "B" (withdraw "D")))

  (check-liquid))
```

During the compilation of the contract, the tool-chain checks if it is liquid. The result is printed before the transactions in a comment-box.

```
/*==========================================================
Model checking result for (check-liquid)

Result: false
counterexample({[0 | nil | 'xconf U empty | empty] < (    A, B) :
     withdraw C + (A, B) : withdraw D, 100000000 satoshi > 'xconf,
         'C-LockAuthControl} {{A lock withdraw C in 'xconf}[0 | nil | '
              xconf U empty    | empty] < Lock((A, B) : withdraw C) + (
              A, B) : withdraw D, 100000000 satoshi >
         'xconf,'Rifl} {{A lock withdraw D in 'xconf}[0 | nil | 'xconf
              U empty |    empty] < Lock((A, B) : withdraw C) + Lock((A
              , B) : withdraw D), 100000000    satoshi > 'xconf,'
              Finalize}, {[0 | nil | 'xconf U empty | empty] < Lock((A,
               B)    : withdraw C) + Lock((A, B) : withdraw D),
              100000000 satoshi > 'xconf,    solution})
Model checking time: 143.0 ms
==============================================================*/
```

The output indicates that the contract is not liquid. In fact, In order to unlock the funds, "A" and "B" must agree on the recipient of the donation, by giving their authorization on the same branch. This contract would be

liquid only by assuming the cooperation between "A" and "B": indeed, "A
" alone cannot guarantee that the 2 ₿ will eventually be donated, as "B" can choose a different recipient, or even refuse to give any authorization.

A possible way to modify the contract to handle this situations by adding a timeout branch with (after 700000 (split (1 -> (withdraw "A"))(1 -> (withdraw "B")))). The new branch locks the contract until the block number 700000 is appended to the blockchain, modelling a delay. After the corresponding time passes, it unlocks and returns their deposits to "A" and "B".

```
(participant "A" "0339bd7fade9167e09681...2bde1d57247fbe1")
(participant "B" "034a7192e922118173906...da4f85701a5f809")
(participant "C" "034f5ca30056b9dd89132...fea9c3467631e6b")
(participant "D" "037b60c121050e1fa6e7d...da1c63f0e2a157e")

(debug-mode)

(contract
  (pre
    (deposit "A" 1 "txid:2e647d8566f00a08d27...b2965e35@1")
    (deposit "A" 1 "txid:625bc69c467b33e2abj...bc710168@0"))

  (choice
    (auth "A" "B" (withdraw "C"))
    (auth "A" "B" (withdraw "D")))
    (after 700000 (split (1 -> (withdraw "A")) (1 -> (withdraw "B"))))
        )

  (check-liquid))
```

Now the contract is liquid, and the toolchain confirms it.

```
/*============================================================
Model checking result for (check-liquid)

Result: true
Model checking time: 322.0 ms
==============================================================*/
```

Liquidity with strategies In the following contract, "A" can reveal her secret and redeem its deposit. Otherwise, after a certain amount of time the block number 700000 will be appended to the blockchain, "B" can redeem "A"'s deposit, after providing his authorization to do so.

```
(participant "A" "0339bd7fade9167e09681...2bde1d57247fbe1")
(participant "B" "034a7192e922118173906...da4f85701a5f809")
(participant "C" "034f5ca30056b9dd89132...fea9c3467631e6b")
(participant "D" "037b60c121050e1fa6e7d...da1c63f0e2a157e")

(debug-mode)
```

```
(contract
  (pre
    (deposit "A" 1 "txid:2e647d8566f00a08d27...b2965e35@1")
    (secret "A" a "f9292914bfd27c426a23465fc122322abbdb63b7"))

  (choice
    (reveal (a) (withdraw "A"))
    (auth "B" (after 700000 (withdraw "B"))))

(check-liquid))
```

Below, we check the strategy-less liquidity, detecting that the contract is not liquid. This is because if neither `"A"` reveals her secret nor `"B"` gives his authorization, the funds will be stuck forever.

```
/*==========================================================
  Model checking result for (check-liquid)

Result: false
Secrets: a:1

counterexample({[0 | 700000 | 'xconf U empty | B, A] <    B : after
     700000 : withdraw B + put empty reveal a if True . withdraw A,
        100000000 satoshi > 'xconf | {A : a # 1},'C-LockAuthRev} {{A
     lock-reveal a}[0 |    700000 | 'xconf U empty | B, A] Lock({A : a
      # 1}) | < B : after 700000 :    withdraw B + put empty reveal a
     if True . withdraw A, 100000000 satoshi >    'xconf,'Rifl} {{B
     lock after 700000 : withdraw B in 'xconf}[0 | 700000 |    'xconf
     U empty | B, A] Lock({A : a # 1}) | < Lock(B : after 700000 :
     withdraw B) + put empty reveal a if True . withdraw A, 100000000
     satoshi >    'xconf,'Rifl} {{delta 700000}[700000 | nil | 'xconf
     U empty | B, A] Lock({A    : a # 1}) | < Lock(B : after 700000 :
     withdraw B) + put empty reveal a if    True . withdraw A,
     100000000 satoshi > 'xconf,'Finalize}, {[700000 | nil |    'xconf
      U empty | B, A] Lock({A : a # 1}) | < Lock(B : after 700000 :
        withdraw B) + put empty reveal a if True . withdraw A,
     100000000 satoshi >    'xconf,solution})
Model checking time: 104.0 ms
==============================================================*/
```

The BitML toolchain allows us to specify the intended behaviour of a participant, called *strategy*. The security propriety is verified with respect to the specified strategies.

We check if the contract is liquid if the strategy of `"A"` consists in revealing her secret, expressed by `(strategy "A"(do-reveal a))` as parameter of `(check-liquid Strategy ...)`.

We also check the liquidity if `"A"` authorizes the second branch of the contract, with the strategy `(strategy "B"(do-auth (auth "B"(after 700000 (withdraw "B"))))`.

```
(participant "A" "0339bd7fade9167e09681...2bde1d57247fbe1")
(participant "B" "034a7192e922118173906...da4f85701a5f809")
(participant "C" "034f5ca30056b9dd89132...fea9c3467631e6b")
(participant "D" "037b60c121050e1fa6e7d...da1c63f0e2a157e")

(debug-mode)

(contract
  (pre
    (deposit "A" 1 "txid:2e647d8566f00a08d27...b2965e35@1")
    (secret "A" a "f9292914bfd27c426a23465fc122322abbdb63b7"))

  (choice
    (reveal (a) (withdraw "A"))
    (auth "B" (after 700000 (withdraw "B"))))

  (check-liquid
    (strategy "A" (do-reveal a)))

  (check-liquid
    (strategy "B" (do-auth (auth "B" (after 700000 (withdraw "B"))))))
        )
```

For both strategies, the contract is liquid.

```
/*==============================================================
Model checking result for (check-liquid (strategy A (do-reveal a)))

Result: true

/*===============================================================
Model checking result for (check-liquid (strategy B (do-auth (auth B (
    after 700000 (withdraw B))))))

Result: true
Model checking time: 270.0 ms
==============================================================*/
```

Quantitative liquidity The previous flavours of liquidity require that no funds remain frozen within the contract. However, in some cases a participant could accept the fact that a portion of the funds remain frozen, especially when these funds would be ideally assigned to other participants.

In the following contract, `"A"` and `"B"` put 1₿ each. Each of them will get their own ₿ back if they reveal their secret.

```
(participant "A" "0339bd7fade9167e...5bbb84211fd12bde1d57247fbe1")
(participant "B" "034a7192e9221181...57fc7f403094da4f85701a5f809")

(debug-mode)

(contract
  (pre
    (deposit "A" 1 "txid:2e647d8566f00a08...161c2078963d8deb2965e35@1"
        )
    (deposit "B" 1 "txid:0f795bda36ac661f...69f5fb7ccc4c3d767db9f34@1"
        )
    (secret "A" a "f9292914bfd27c426a23465fc122322abbdb63b7")
    (secret "B" b "9804ebb0fc4a8329981dd33aaff32b6cb579580a"))

  (split
    (1 -> (reveal (a) (withdraw "A")))
    (1 -> (reveal (b) (withdraw "B"))))

  (check "A" has-more-than 1
    (strategy "A" (do-reveal a)))))
```

In this setting, "A" is interested in checking if she will get back her bitcoin, assuming that she reveals her secret. We check it using `(check "A" has-more-than 1 (strategy "A"(do-reveal a)))`.

```
/*==========================================================
Model checking result for (check A has-more-than 1 (strategy A (do-
    reveal a)))

Result: true
Model checking time: 134.0 ms
==============================================================*/
```

Custom LTL queries The following contract is a *timed commitment*, where "A" wants to choose a secret a, and reveal it before the deadline d; if "A" does not reveal the secret within d, "B" can redeem the 1 ₿ deposit as a compensation.

```
(participant "A" "029c5f6f5ef0095f547799...289fbc90c70209c1cced")
(participant "B" "022c3afb0b654d3c2b0e2f...2f86f7647fa7b817af30")

(debug-mode)

(define (d) 700000)

(contract
  (pre
    (deposit "A" 1 "txA@0")
    (secret "A" a "f9292914bfd27c426a23465fc122322abbdb63b7"))

  (choice
    (reveal (a) (withdraw "A"))
    (after (ref (d)) (withdraw "B")))

  (check-query
    "[]<> (a revealed => A has-deposit>= 100000000 satoshi)")

  (check-query
    "[]<> (a revealed \\/ B has-deposit>= 100000000 satoshi)"))
```

The toolchain allows us to check custom LTL properties, tailored specifically for the contract being verified, using `(check-query "query")`.

In the timed commitment contract, we want the following two properties to be satisfied.

- If `"A"` reveal her secret, she will get back her deposit. We check this property with `(check-query "[]<> (a revealed => A has-deposit>= 100000000 satoshi)")`.
- Either `"B"` gets to know the secret, or he will get the bitcoin as compensation. We check this property with `(check-query "[]<> (a revealed \\/ B has-deposit>= 100000000 satoshi)"))`.

Due to the internal representation of numbers in the model checker, all ₿ values have to be expressed in *satoshi* when checking custom LTL queries.

The result is true for both queries:

```
/*============================================================
Model checking result for (check-query [] (a revealed => <> A has-
    deposit>= 100000000 satoshi))

Result: true

/*============================================================
Model checking result for (check-query []<> (a revealed \/ B has-
    deposit>= 100000000 satoshi))

Result: true
Model checking time: 408.0 ms
==============================================================*/
```

The first LTL property has the same semantic as checking the quantitative liquidity of 1 ₿ if the strategy of `"A"` is to reveal her secret, or `(check "A" has-more-than 1 (strategy "A"(do-reveal a)))`. Instead, the second LTL property cannot be expressed as a combination of liquidity and strategies.

Other that `revealed` and `has-deposit>=`, you can express your LTL properties with `has-deposit`, and `has-deposit<=`.

Chapter 7

On-chain Fungible Tokens on Bitcoin

One of the main applications of blockchain technologies is the exchange of custom crypto-assets, called *tokens*. Token transfers currently involve $\sim 50\%$ of the transactions on the Ethereum blockchain [107], and they are at the basis of many protocols built on top of that platform [6, 44]. Broadly, tokens are classified as *fungible* or *non-fungible*. Fungible tokens can be split into smaller units: different units of the same token can be used interchangeably. Further, users can join units of the same fungible token, and exchange them with other crypto-assets. Instead, non-fungible tokens cannot be split or joined.

Historically, the first implementations of tokens were developed before Ethereum, on top of Bitcoin. Some of them (e.g., [105]) used small bitcoin fractions to represent the token value; some others (e.g., [118, 100, 103]) embedded the token value in other transaction fields [13], to cope with the fluctuating bitcoin price. All these implementations have a common drawback: the correctness of the token actions is *not* guaranteed by the consensus protocol of the blockchain. In fact, the blockchain is used just to notarize the actions that manipulate tokens, but not to check that these actions are actually permitted. Typically, the owners of these tokens must resort to *off-chain* mechanisms (e.g., trusted authorities) to have some guarantees on the correct use of tokens, e.g. that they are not double-spent, or that distinct tokens are not joined.

By contrast, modern blockchain platforms support *on-chain* tokens, whose correctness is guaranteed by the consensus protocol of the blockchain. Some platforms (e.g., Algorand [79]) natively support tokens, while some others (e.g., Ethereum) encode them as smart contracts. Bitcoin, instead, does not support tokens natively, and its limited script language is not expressive enough to implement them as smart contracts. Since adding native tokens to Bitcoin appears to be out of reach, given the resilience of the Bitcoin community to radical changes [102], the only viable alternative is to devise a small, efficient extension of the script language which increases the expressiveness of Bitcoin enough to support tokens.

We implement a token contract as a single, succinct script which exploits neighbourhood covenants. We define a symbolic model of token actions, and a computational model, where performing these actions corresponds to appending transactions to the Bitcoin blockchain.

7.1 Overview of the approach

In this section we summarize our approach: in particular, we sketch our implementation of Bitcoin tokens, motivating the use of neighbourhood covenants to guarantee their security.

Tokens We propose a symbolic model of fungible tokens. Since non-fungible tokens are the special case of fungible ones where each token is generated exactly in one unit, hereafter we consider the general case of fungible tokens. The basic element of our model is the *deposit*, i.e. a term of the form:

$$\langle \mathsf{A}, v : \tau \rangle_x \qquad (v \in \mathbb{N})$$

which represents the fact that a user A owns v units of a token τ, where τ may denote either user-defined tokens or bitcoins (₿). The index x uniquely identifies the term within a *configuration*, i.e. a composition of deposits, e.g.:

$$\langle \mathsf{A}, 1 : \tau \rangle_x \mid \langle \mathsf{A}, 2 : \tau \rangle_y \mid \langle \mathsf{B}, 3 : ₿ \rangle_z$$

We define a few actions to mint and manipulate tokens. First, any user A can mint v units of a new token, spending a deposit of 0 ₿. Performing this action (say, with $v = 10$) is modelled as a state transition, whose labels records the performed action:

$$\langle \mathsf{A}, 0 : ₿ \rangle_{x_0} \xrightarrow{gen} \langle \mathsf{A}, 10 : \tau \rangle_{x_1} \tag{7.1}$$

where the identifier x_1 of the new deposit and the identifier τ of the minted token are *fresh*. After performing the action, A owns a deposit of ten units of the token τ. As said before, one of the peculiar properties of fungible tokens is that they can be *split*. When splitting her deposit in two smaller deposits, A can choose the owner of one of the new deposits, e.g.:

$$\langle \mathsf{A}, 10 : \tau \rangle_{x_1} \xrightarrow{split} \langle \mathsf{A}, 8 : \tau \rangle_{x_2} \mid \langle \mathsf{B}, 2 : \tau \rangle_{x_3} \tag{7.2}$$

A user can transfer the ownership of any of her deposits to another user. For instance, A can *give* her deposit x_2 to B:

$$\langle \mathsf{A}, 8 : \tau \rangle_{x_2} \xrightarrow{give} \langle \mathsf{B}, 8 : \tau \rangle_{x_4} \tag{7.3}$$

After that, B owns a total of 10 units of τ in two separate deposits, one with 8 units, and the other one with 2 units. This reflects the UTXO nature of Bitcoin: by contrast, in account-based blockchains like Ethereum, B would have a single account storing 10 units of τ. Now, B can *join* his two deposits, obtaining a single deposit with 10 units of τ. When performing the *join* action, B can also choose the owner of the new deposit, in this case transferring it back to A:

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{B}, 2 : \tau \rangle_{x_3} \xrightarrow{join} \langle \mathsf{A}, 10 : \tau \rangle_{x_5} \tag{7.4}$$

A crucial property of the *join* operation is that only deposits of the same token can be joined together. Thus, two deposits of τ and τ' with $\tau \neq \tau'$ cannot be joined:

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{A}, 2 : \tau' \rangle_{x_6} \not\xrightarrow{join}$$

In this configuration, if both A and B agree, they can *exchange* the ownership of their tokens:

$$\langle \mathsf{B}, 8 : \tau \rangle_{x_4} \mid \langle \mathsf{A}, 2 : \tau' \rangle_{x_6} \xrightarrow{xchg} \langle \mathsf{A}, 8 : \tau \rangle_{x_7} \mid \langle \mathsf{B}, 2 : \tau' \rangle_{x_8}$$

The *xchg* operation also supports the exchange between bitcoins and other tokens, representing the trade of tokens. For instance, A can buy 2 units of τ' from B for 1₿:

$$\langle \mathsf{B}, 2 : \tau' \rangle_{x_8} \mid \langle \mathsf{A}, 1 : ₿ \rangle_{x_9} \xrightarrow{xchg} \langle \mathsf{A}, 2 : \tau' \rangle_{x_{10}} \mid \langle \mathsf{B}, 1 : ₿ \rangle_{x_{11}}$$

Bitcoin Although Bitcoin does not support user-defined tokens, it implements all the operations discussed above on its native crypto-currency. Intuitively, each deposit corresponds to a transaction output, and performing actions corresponds to appending a suitable transaction that redeems it.

For instance, minting bitcoins is obtained through coinbase transactions, which are used in Bitcoin to pay rewards to miners. We represent a coinbase transaction as follows:

$\mathsf{T}_{7.1}$
$\mathsf{in}(1)$: $\perp$
$\mathsf{wit}(1)$: $\perp$
$\mathsf{out}(1)$: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{A}}, \mathsf{rtx}.\mathsf{wit}),\ \mathsf{val} : 10\text{₿}\}$

In general, the in field points to a previous transaction on the blockchain, that the current one is trying to *spend*. Here, the "undefined" value $\perp$ characterizes $\mathsf{T}_{7.1}$ as a coinbase, since it mints bitcoins without spending any transaction. The out field is a record, where scr is a *script*, and val is the amount of bitcoins that will be redeemed by a subsequent transaction which points to $\mathsf{T}_{7.1}$ and satisfies its script. Here, the script $\mathsf{versig}(pk_{\mathsf{A}}, \mathsf{rtx}.\mathsf{wit})$ verifies a signature on the redeeming transaction (rtx, excluding its wit field) against A's public key pk_{A}. This signature is retrieved from the wit field of rtx. Since A is the only user who can redeem $\mathsf{T}_{7.1}$, we can say that $\mathsf{T}_{7.1}$ is the *computational counterpart* of the deposit $\langle \mathsf{A}, 10 : \text{₿}\rangle_{x_1}$.

To perform the *split* action (7.2) on $\tau = \text{₿}$, we can spend $\mathsf{T}_{7.1}$ with a transaction $\mathsf{T}_{7.2}$ with *two* outputs:

$\mathsf{T}_{7.2}$
$\mathsf{in}(1)$: $(\mathsf{T}_{7.1}, 1)$
$\mathsf{wit}(1)$: $sig_{sk_{\mathsf{A}}}(\mathsf{T}_1)$
$\mathsf{out}(1)$: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{A}}, \mathsf{rtx}.\mathsf{wit}),\ \mathsf{val} : 8\text{₿}\}$
$\mathsf{out}(2)$: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{B}}, \mathsf{rtx}.\mathsf{wit}),\ \mathsf{val} : 2\text{₿}\}$

The first output, that we denote by $(\mathsf{T}_{7.2}, 1)$, corresponds to the deposit $\langle \mathsf{A}, 8 : \text{₿}\rangle_{x_2}$ in (7.2). Instead, the output $(\mathsf{T}_{7.2}, 2)$ corresponds to $\langle \mathsf{B}, 2 : \text{₿}\rangle_{x_3}$. These outputs can be spent independently. For instance, performing the *give* action in (7.3) corresponds to appending a transaction which spends $(\mathsf{T}_{7.2}, 1)$:

$\mathsf{T}_{7.3}$
$\mathsf{in}(1)$: $(\mathsf{T}_{7.2}, 1)$
$\mathsf{wit}(1)$: $sig_{sk_{\mathsf{A}}}(\mathsf{T}_{7.3})$
$\mathsf{out}(1)$: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{B}}, \mathsf{rtx}.\mathsf{wit}),\ \mathsf{val} : 8\text{₿}\}$

At this point, we have two unspent outputs on the blockchain: $(\mathsf{T}_{7.2}, 2)$ and $(\mathsf{T}_{7.3}, 1)$. We can perform the *join* action in (7.4) by spending both of them *simultaneously* with the following transaction, which has *two* inputs:

$\mathsf{T}_{7.4}$	
$\mathsf{in}(1)$: $(\mathsf{T}_{7.2}, 2)$	$\mathsf{in}(2)$: $(\mathsf{T}_{7.3}, 1)$
$\mathsf{wit}(1)$: $sig_{sk_{\mathsf{B}}}(\mathsf{T}_{7.4})$	$\mathsf{wit}(2)$: $sig_{sk_{\mathsf{B}}}(\mathsf{T}_{7.4})$
out: $\{\mathsf{scr} : \mathsf{versig}(pk_{\mathsf{B}}, \mathsf{rtx}.\mathsf{wit}), \mathsf{val} : 10₿\}$	

Implementing Bitcoin tokens with covenants Although the Bitcoin script language is a bit more flexible than shown above, it does not allow to implement *on-chain* tokens. One of the first techniques to embed on-chain tokens in an *extended* version of Bitcoin was described in [63]. The technique relies on *covenants*, an extension of Bitcoin scripts which allows transactions to constrain the scripts of the redeeming ones. Roughly, a transaction output containing the script:

$$e \text{ and } \mathsf{verrec}(\mathsf{rtxo}(n))$$

where e is an arbitrary script, can only be redeemed by a transaction which makes e evaluate to true, and whose script in the n-th output is syntactically equal to e and $\mathsf{verrec}(\mathsf{rtxo}(n))$. Using covenants, we can mint a token by appending the transaction T below, where the extra field arg is syntactic sugar for a sequence of values accessible by the script:

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

The arg field identifies A as the owner of the token: to transfer the ownership to B, A must spend T with a transaction T', setting its arg to B's public key. For this to be possible, T' must satisfy the conditions specified in T's script: (i) the wit field must contain the signature of the current owner; (ii) the output at index 1 must have 1₿ value, to preserve the value of the token; (iii) the script at index 1 in T' must be equal to that in T. Once T' is on the blockchain, B can transfer the token to another user, by appending a transaction which redeems T'.

Note that the transaction T above actually mints a *non*-fungible token, which can be transferred from one user to another, but whose value cannot be *split* (further, the token has a subtle flaw related to *join* actions: we will say more on this). The first step to turn the token into a fungible one is to support the *split* action. We can achieve this by adding a second element to the arg sequence, to represent the number of token units deposited in the transaction output. Using the notation $w.i$ to access the i-th element of a sequence w (for $1 \leq i \leq |w|$), we can implement a splittable token as follows:

```
$\mathsf{T}_{split}$
$\cdots$
$\mathsf{out}(1)$: $\{\mathsf{arg} : pk_{\mathsf{A}}\ v,$
          $\mathsf{scr} : \mathsf{versig}(\mathsf{ctxo.arg}.1, \mathsf{rtx.wit})$ and
               $\mathsf{rtxo}(1).\mathsf{arg}.2 + \mathsf{rtxo}(2).\mathsf{arg}.2 = \mathsf{ctxo.arg}.2$ and
               $\mathsf{verrec}(\mathsf{rtxo}(1))$ and $\mathsf{verrec}(\mathsf{rtxo}(2))$ and
               $\mathsf{outlen}(\mathsf{rtx}) = 2$
          $\mathsf{val} : \cdots\}$
```

The last two lines of the script ensure that any transaction which redeems T_{split} has exactly two outputs, each one with the same script of T_{split}. The second line ensures that the *split* preserves the number of token units (here, val is immaterial).

Now, let e_{split} be the script used in T_{split}. To extend the token with the *join* action, first we need to add a third element to the arg sequence, to encode the action performed by a transaction (say, G for *gen*, S for *split*, and J for *join*). The extended script could have the following form:

$$e \triangleq \mathsf{if}\ \mathsf{rtxo}(1).\mathsf{arg}.3 = S\ \mathsf{then}\ e_{split}\ \mathsf{else}\ e_{join}$$

where e_{join} implements the join functionality, i.e.: (i) verify the signature on the redeeming transaction; (ii) check that the redeeming transaction has exactly *two* inputs and one output; (iii) ensure that the token units are preserved; (iv) ensure that the joined transactions represent units of the *same* token.

For instance, consider the transactions in Figure 7.1, where T_4 and T_3 represent, respectively, the deposits $\langle \mathsf{B}, 8 : \tau \rangle_{x_4}$ and $\langle \mathsf{B}, 2 : \tau \rangle_{x_3}$. To perform the *join* action in (7.4), we must spend T_4 and T_3 with the transaction T_5: this requires to satisfy the script e in T_4 and T_3. For condition (iii), the script must ensure that the 10 token units redeemed by T_5 are the sum of the 8 units in T_4 and the 2 units in T_3. For condition (iv), the script in T_4 should check that it is the same as that in T_3, and *viceversa*. Hence, to implement conditions (iii)-(iv), the script in a transaction output must be able to access the fields in its *sibling*, i.e. the transaction output which is redeemed together (e.g., $(\mathsf{T}_3, 1)$ is the sibling

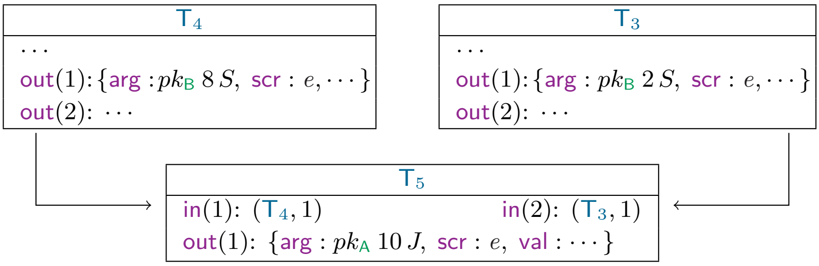

T_4
$\cdots$
$\mathsf{out}(1)\!:\{\mathsf{arg}:pk_{\mathsf{B}}\ 8\,S,\ \mathsf{scr}:e,\cdots\}$
$\mathsf{out}(2)\!:\ \cdots$
T_3
$\cdots$
$\mathsf{out}(1)\!:\{\mathsf{arg}:pk_{\mathsf{B}}\ 2\,S,\ \mathsf{scr}:e,\cdots\}$
$\mathsf{out}(2)\!:\ \cdots$
T_5
$\mathsf{in}(1)\!:\ (\mathsf{T}_4,1)$
$\mathsf{in}(2)\!:\ (\mathsf{T}_3,1)$
$\mathsf{out}(1)\!:\ \{\mathsf{arg}:pk_{\mathsf{A}}\ 10\,J,\ \mathsf{scr}:e,\ \mathsf{val}:\cdots\}$

Figure 7.1: *A transaction T_5 attempting to join T_4 and T_3.*

of $(\mathsf{T}_4, 1)$ when appending T_5). However, neither Bitcoin nor its extensions with covenants [18, 63, 66, 70] allow scripts to access the siblings.

An insecure implementation of join To implement the *join* action, we start by extending Bitcoin scripts with an operator to access the sibling transaction outputs:

$$\mathsf{stxo}(n) \triangleq \text{output redeemed by the } n\text{-th input of } \mathsf{rtx}$$

Using this new operator, we can encode the conditions (iii) and (iv) in e_{join} as follows:

$$\mathsf{rtxo}(1).\mathsf{arg}.2 = \mathsf{stxo}(1).\mathsf{arg}.2 + \mathsf{stxo}(2).\mathsf{arg}.2 \qquad \text{(iii)}$$
$$\mathsf{verrec}(\mathsf{stxo}(1)) \ \mathsf{and}\ \mathsf{verrec}(\mathsf{stxo}(2)) \qquad \text{(iv)}$$

Although this implementation of e_{join} correctly encodes the conditions, it introduces a security vulnerability: an adversary can join two deposits of *different* tokens. The attack is exemplified in Figure 7.2. The transactions T_{A} and T_{M} mint 10 units of *different* tokens, and transaction T_3 joins them into a single deposit of the *same* token. Ideally, to counter this attack, e_{join} should check not only the sibling, but also its ancestors until the minting transaction, and verify that it corresponds to the minting ancestor of the current transaction output. Although this would be possible by adding script operators that can go up the transaction graph at an arbitrary depth, this would be highly inefficient from the point of view of miners, who should record the *whole* transaction graph, instead of just the set of unspent transactions (UTXO).

A secure implementation with neighbourhood covenants To address this issue, we use an operator which can go up the transaction graph

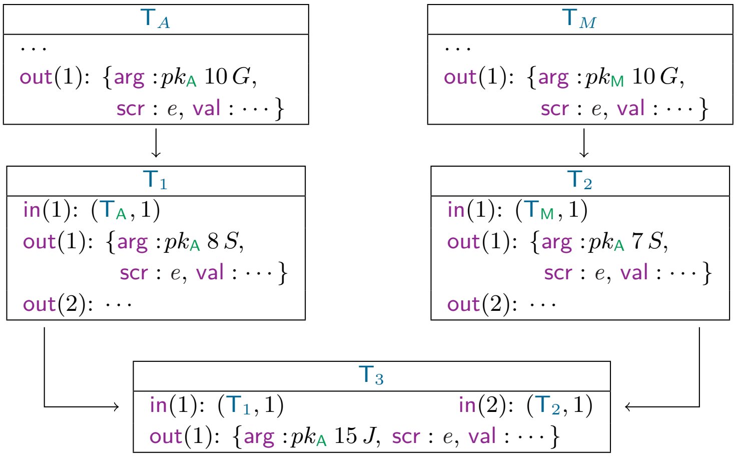

T_A
· · ·
out(1): {arg : pk_A 10 G, scr : e, val : · · ·}
T_M
· · ·
out(1): {arg : pk_M 10 G, scr : e, val : · · ·}
T_1
in(1): (T_A, 1)
out(1): {arg : pk_A 8 S, scr : e, val : · · ·}
out(2): · · ·
T_2
in(1): (T_M, 1)
out(1): {arg : pk_A 7 S, scr : e, val : · · ·}
out(2): · · ·
T_3
in(1): (T_1, 1)
in(2): (T_2, 1)
out(1): {arg : pk_A 15 J, scr : e, val : · · ·}

Figure 7.2: *A join attack merging two different tokens.*

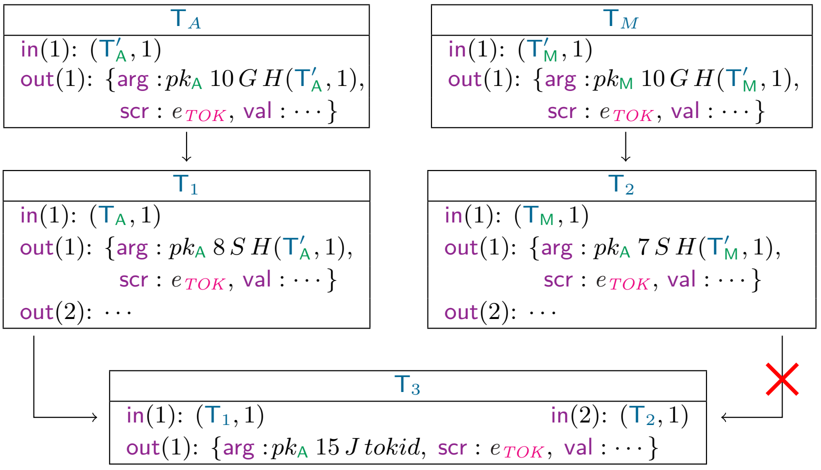
only one level, introduced by neighborhood covenants in Chapter 4. By exploiting this new covenant, we can thwart the *join* attack of Figure 7.2, and eventually obtain a secure and efficient implementation of fungible tokens. We now sketch the script e_{TOK} which implements tokens. First, we add a fourth element to the arg sequence, to record in each transaction output the identifier of the token deposited in that output. As an identifier, we use the hash of the *parent* of the minting transaction, which we access through the script txid(ptxo(1)). When we evaluate the script contained in the minting transaction (e.g., T_{A} and T_{M} in Figure 7.2), we require that its arg field actually contains the token identifier:

$$e_{gen} \triangleq \cdots \ \text{and ctxo.arg}.4 = \text{txid}(\text{ptxo}(1))$$

Then, in the sub-scripts corresponding to all other token actions, we check that the redeeming transaction preserves the token identifier. E.g., in the *join* sub-script, besides checking conditions (iii) and (iv) as shown before, we add the condition:

$$e_{join} \triangleq \cdots \ \text{and ctxo.arg}.4 = \text{rtxo}(1).\text{arg}.4$$

In Figure 7.3 we show how this resolves the attack of Figure 7.2. In order to append the malicious *join* transaction T_3, we must satisfy the script e_{TOK} in both T_1 and T_2. These scripts check that the *tokid* in the redeeming transaction T_3 is equal to the identifiers of the two branches,



T_A
in(1): $(\mathsf{T}'_\mathsf{A}, 1)$
out(1): {arg : pk_A $10\,G\,H(\mathsf{T}'_\mathsf{A}, 1)$, scr : e_{TOK}, val : ⋯}
T_M
in(1): $(\mathsf{T}'_\mathsf{M}, 1)$
out(1): {arg : pk_M $10\,G\,H(\mathsf{T}'_\mathsf{M}, 1)$, scr : e_{TOK}, val : ⋯}
T_1
in(1): $(\mathsf{T}_\mathsf{A}, 1)$
out(1): {arg : pk_A $8\,S\,H(\mathsf{T}'_\mathsf{A}, 1)$, scr : e_{TOK}, val : ⋯}
out(2): ⋯
T_2
in(1): $(\mathsf{T}_\mathsf{M}, 1)$
out(1): {arg : pk_A $7\,S\,H(\mathsf{T}'_\mathsf{M}, 1)$, scr : e_{TOK}, val : ⋯}
out(2): ⋯
T_3
in(1): $(\mathsf{T}_1, 1)$
in(2): $(\mathsf{T}_2, 1)$
out(1): {arg : pk_A $15\,J\,tokid$, scr : e_{TOK}, val : ⋯}

Figure 7.3: *Thwarting the join attack.*

$H(\mathsf{T}'_\mathsf{A}, 1)$ and $H(\mathsf{T}'_\mathsf{M}, 1)$: by collision resistance of the hash function, this is not possible.

The discussion above shows how to counter an attack which attempts to *join* different tokens. However, the adversary could devise more ingenious attacks, e.g. forging units of an existing token. For instance, if T_M in Figure 7.3 were storing ₿, its owner could spend it to create a transaction T_2 with *arbitrary* scripts and arguments. In particular, T_2 could use e_{TOK} and any token identifier in arg.4, e.g., $H(\mathsf{T}'_\mathsf{A}, 1)$, effectively *forging* new units of a pre-existing token. Our full e_{TOK} script exploits neighbourhood covenants to prevent these kinds of attacks as well.

7.2 A symbolic model of tokens

Let $\mathsf{A}, \mathsf{B}, \ldots$ range over *users*, and let $\tau, \tau', \ldots$ range over *tokens*, encompassing both user-defined ones and bitcoins (₿). A term $\langle \mathsf{A}, v : \tau \rangle_x$ represents a *deposit* of $v \in \mathbb{N}$ units of the token τ owned by A (the index x is an unique identifier of the deposit). A term $\mathsf{A} \rhd_x \alpha$ represents A's *authorization* to perform the *action* α on the deposit x. The possible token actions are the following:

- $gen(x, v)$ represents the act of spending a bitcoin deposit x to mint v units of a new token. The owner of these units is the user who owned the deposit x.

- $\mathit{burn}(\boldsymbol{x}, y)$ represents the act of destroying a sequence of deposits $\boldsymbol{x}$, moving them to an unspendable deposit y.
- $\mathit{split}(x, v, \mathsf{B})$ represents the act of splitting a deposit x (say, containing $v + v'$ units of a token τ) in two deposits of τ. The first one of these deposits is owned by the same owner of x, and contains v token units. The second one if owned by B, and contains the remaining v' units.
- $\mathit{join}(x, y, \mathsf{C})$ represents the joining of two deposits x and y of the same token, into a new deposit, owned by C.
- $\mathit{xchg}(x, y)$ represents the act of atomically exchanging the owners of the two deposits x and y (not both of bitcoins). In particular, when one of the deposits stores ₿, this action represents buying/selling tokens for bitcoins.
- $\mathit{give}(x, \mathsf{B})$ represents a donation of the deposit x to B.

Users follow a common pattern to perform token actions: (i) first, the involved users grant their authorization on the action; (ii) once all the needed authorizations have been granted, the action can actually be performed.

A *configuration* Γ is a compositions of deposits and authorizations. We assume that configurations form a commutative monoid under the composition operator $|$, and we use $\mathbf{0}$ to denote the empty configuration. We require that if $\langle \mathsf{A}, v : \tau \rangle_x$ and $\langle \mathsf{B}, v' : \tau' \rangle_{x'}$ both occur in Γ, then $x \neq x'$. We define a transition semantics between configurations in Figure 7.4. Transitions are decorated with labels, which describe the performed actions.

Rule [Gen] consumes a bitcoin deposit x owned by A to generate v units of a new token τ, which are stored in a fresh deposit y. Note that A's authorization is required to perform the action. For simplicity, we assume that minting tokens has no cost: it would be straightforward to adapt the rule to require a minting fee. Rule [Burn] removes from the configuration a single token deposit (when $n = 1$ and $\tau_1 \neq$ ₿), or atomically removes a sequence of bitcoin deposits (when $\tau_i =$ ₿ for all i). Rule [Split] divides a deposit x in two fresh deposits y and z, preserving the number of token units. Rule [Join] allows A and B to merge two deposits of a token τ, preserving the amount of token units, and transferring the new deposit to C. Rule [Xchg] allows A and B to swap two deposits, containing either user-defined tokens or bitcoins. Finally, rule [Give] allows A to donate one of her deposits to another user.

The transition relation $\rightarrow$ is non-deterministic, because of the fresh

$$\frac{\Gamma = \mathsf{A} \rhd_x \mathit{gen}(x,v) \mid \Gamma' \quad v > 0 \quad y, \tau \text{ fresh}}{\langle \mathsf{A}, 0 : \text{₿} \rangle_x \mid \Gamma \xrightarrow{\mathit{gen}(x,v)} \langle \mathsf{A}, v : \tau \rangle_y \mid \Gamma'} \text{[Gen]}$$

$$\frac{\Gamma = \left(\|_{i \in 1..n} \mathsf{A}_i \rhd_{x_i} \mathit{burn}(x_1 \cdots x_n, y) \right) \mid \Gamma' \quad n = 1 \lor (n \geq 1 \land \forall i : \tau_i = \text{₿})}{\left(\|_{i \in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \mid \Gamma \xrightarrow{\mathit{burn}(x_1 \cdots x_n, y)} \Gamma'} \text{[Burn]}$$

$$\frac{\Gamma = \mathsf{A} \rhd_x \mathit{split}(x,v,\mathsf{B}) \mid \Gamma' \quad v, v' \geq 0 \quad y, y' \text{ fresh}}{\langle \mathsf{A}, (v+v') : \tau \rangle_x \mid \Gamma \xrightarrow{\mathit{split}(x,v,\mathsf{B})} \langle \mathsf{A}, v : \tau \rangle_y \mid \langle \mathsf{B}, v' : \tau \rangle_{y'} \mid \Gamma'} \text{[Split]}$$

$$\frac{\Gamma = \mathsf{A} \rhd_x \mathit{join}(x,y,\mathsf{C}) \mid \mathsf{B} \rhd_y \mathit{join}(x,y,\mathsf{C}) \mid \Gamma' \quad z \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau \rangle_y \mid \Gamma \xrightarrow{\mathit{join}(x,y,\mathsf{C})} \langle \mathsf{C}, (v+v') : \tau \rangle_z \mid \Gamma'} \text{[Join]}$$

$$\frac{\Gamma = \mathsf{A} \rhd_x \mathit{xchg}(x,y) \mid \mathsf{B} \rhd_y \mathit{xchg}(x,y) \mid \Gamma' \quad \tau \neq \text{₿} \quad x', y' \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau' \rangle_y \mid \Gamma \xrightarrow{\mathit{xchg}(x,y)} \langle \mathsf{A}, v' : \tau' \rangle_{x'} \mid \langle \mathsf{B}, v : \tau \rangle_{y'} \mid \Gamma'} \text{[Xchg]}$$

$$\frac{\Gamma = \mathsf{A} \rhd_x \mathit{give}(x,\mathsf{B}) \mid \Gamma' \quad y \text{ fresh}}{\langle \mathsf{A}, v : \tau \rangle_x \mid \Gamma \xrightarrow{\mathit{give}(x,\mathsf{B})} \langle \mathsf{B}, v : \tau \rangle_y \mid \Gamma'} \text{[Give]}$$

Figure 7.4: *Semantics of token actions.*

names generated for deposits and tokens: however, given a transition $\Gamma \xrightarrow{\alpha} \Gamma'$ the label α is uniquely determined from Γ and Γ'. A *symbolic run* $\mathcal{S}$ is a (possibly infinite) sequence $\Gamma_0 \Gamma_1 \cdots$, where Γ_0 contains only ₿ deposits, and for all $i \geq 0$ there exists some (unique) α_i such that $\Gamma_i \xrightarrow{\alpha_i} \Gamma_{i+1}$. For all $i \geq 0$, we denote with $\mathcal{S}_i$ the i-th element of the run, when this element exists. If $\mathcal{S}$ is finite, we denote its length as $|\mathcal{S}|$, and we write $\Gamma_{\mathcal{S}}$ for its last configuration, i.e. $\mathcal{S}_{|\mathcal{S}|-1}$.

Definition 7.1 (Token balance)**.** *We define the balance of a token $\tau \neq$ ₿ in a configuration Γ inductively as follows:*

$$\begin{array}{ll} \mathit{bal}_\tau(\mathbf{0}) = 0 & \mathit{bal}_\tau(\Gamma \mid \Gamma') = \mathit{bal}_\tau(\Gamma) + \mathit{bal}_\tau(\Gamma') \\ \mathit{bal}_\tau(\langle \mathsf{A}, v : \tau \rangle_x) = v & \mathit{bal}_\tau(\langle \mathsf{A}, v : \tau' \rangle_x) = 0 \quad (\tau' \neq \tau) \end{array}$$

The following theorem establishes a basic preservation property: the balance of a token after a run is equal to the *minted* value minus the *burnt* value, defined as:

$$\mathit{minted}_\tau(\mathcal{S}) = v \ \text{ if } \exists i : \begin{array}{l} \mathcal{S}_i \xrightarrow{gen(x,v)} \mathcal{S}_{i+1}, \text{ and} \\ \tau \text{ occurs in } \mathcal{S}_{i+1} \text{ but not in } \mathcal{S}_i \end{array}$$

$$\mathit{burnt}_\tau(\mathcal{S}) = \sum \left\{ v \;\middle|\; \exists i : \begin{array}{l} \mathcal{S}_i \xrightarrow{burn(x,y)} \mathcal{S}_{i+1}, \text{ and} \\ \mathcal{S}_i = \Gamma \mid \langle \mathsf{A}, v : \tau \rangle_x \end{array} \right\}$$

Lemma 7.2.1. Let $\mathcal{S}$ be a finite symbolic run. For all $\tau \neq$ ₿:

$$\mathit{bal}_\tau(\mathcal{S}) \;=\; \mathit{minted}_\tau(\mathcal{S}) - \mathit{burnt}_\tau(\mathcal{S})$$

Proof. By induction on $\mathcal{S}$, and by case analysis on each step. Inspecting each symbolic semantics rule, we can see that each step preserves the amount of token units, except for minting ([Gen]) and burning ([Burn]), which are explicitly taken into account by the equation. □

7.3 Implementing tokens in Bitcoin

In this section we show how to implement token actions in Bitcoin. To this purpose, we define a *computational model*, which describes the interactions of users who exchange messages and append transactions to the Bitcoin blockchain. A *computational run* $\mathcal{C}$ is a sequence of bitstrings γ, each of which encodes one of the following actions: (i) $\mathsf{A} \rightarrow * : m$, denoting the broadcast of a bitstring m; (ii) T, denoting the appending of a transaction T to the blockchain. A computational run always starts from a coinbase transaction T_0. By extracting the transactions from a run $\mathcal{C}$, we obtain a blockchain $\mathbf{B}_{\mathcal{C}}$.

We simulate each symbolic token action in Figure 7.4 by appending a suitable transaction to the blockchain. We use the arg part of transaction outputs to record the token data:

1. op is the action implemented by the transaction: $0 = \mathit{gen}$, $1 = \mathit{burn}$, $2 = \mathit{split}$, $3 = \mathit{join}$, $4 = \mathit{xchg}$, $5 = \mathit{give}$;
2. owner is the (public key of the) user who owns of the token units controlled by the tx output;
3. tkval is the number of units controlled by the tx output;
4. tkid is the unique token identifier.

Symbolic authorization steps correspond to computational broadcasts of signatures.

We implement the token actions as a single script e_{TOK}, which uses a switch on the op value to jump to the first instruction of the requested action. Since the script is quite complex, we present independently the parts corresponding to each action, postponing the complete script to Appendix 7.4. All the transactions implementing token actions use *exactly* the same script, which we preserve throughout executions by using recursive covenants. To improve readability, we refer to the elements of the arg sequence by name rather than by index, e.g. we write o.op rather than o.arg.1.

Gen We implement the *gen* action by the following script:

```
1 if not verrec(ptxo(1))          // ctxo is a gen
2 then ctxo.tkid = txid(ptxo(1)) // token id
3      and ptxo(1).val = 0        // spent txo has 0 BTC
4      and outlen(ctxo) = 1       // gen has 1 out
5      and ctxo.tkval > 0         // positive token val
6 else ... // the other branches must preserve token id
```

Recall that *gen* produces a symbolic step of the form:

$$\langle \mathsf{A}, 0 : \text{₿} \rangle_x \xrightarrow{gen(x,v)} \langle \mathsf{A}, v : \tau \rangle_y \qquad (v > 0)$$

To translate this symbolic action into a computational one, we must spend a transaction output corresponding to $\langle \mathsf{A}, 0 : \text{₿} \rangle_x$, and produce a fresh output corresponding to $\langle \mathsf{A}, v : \tau \rangle_y$. Assuming that the deposit x corresponds to an unspent transaction output $(\mathsf{T}', 1)$ on the blockchain, this requires to append a transaction T redeeming $(\mathsf{T}', 1)$, and ensuring that:

1. the parent transaction output $(\mathsf{T}', 1)$ is not a token deposit, but just a plain ₿ deposit (line 1);
2. tkid is the identifier of the parent tx output (line 2). This corresponds to identifying the fresh name τ with the deposit name x of the redeemed ₿ deposit;
3. 0₿ are redeemed from the parent transaction (line 3);
4. T has exactly one output (line 4);
5. tkval is positive (line 5), corresponding to the constraint $v > 0$ in the symbolic semantics.

Notice that the first time the script is evaluated is when *redeeming* T (not when appending it). At that time, ctxo will evaluate to $(\mathsf{T}, 1)$, and ptxo(1) to $(\mathsf{T}', 1)$. The script ensures that, when T is redeemed, its tkid will contain a unique identifier of the token. Crucially, the scripts which implement the other token actions will preserve this identifier in the tkid parameter. This identifier is essential to guarantee the correctness of the *join* and *xchg* actions.

Burn We implement the *burn* action by the script:

```
versig(ctxo.owner,rtx.wit) and // check owner
verscr(false,rtxo(1)) and      // make rtx unspendable
inlen(rtxo(1)) = 1 and         // rtx has 1 in
outlen(rtxo(1)) = 1            // rtx has 1 out
```

Recall that *burn* produces a symbolic step:

$$\left(\|_{\, i\in 1..n} \langle \mathsf{A}_i, v_i : \tau_i \rangle_{x_i} \right) \xrightarrow{burn(x_1 \cdots x_n, y)} \mathbf{0}$$

There are two cases, according to whether we are burning a single token deposit, or one or more ₿ deposits. In the first case, assuming that the computational counterpart of x_1 is the output $(\mathsf{T}', 1)$, the corresponding computational step is to append a transaction T redeeming $(\mathsf{T}', 1)$. The witness of T must carry a signature of the owner A, and its output script is *false*, making it unspendable. In the second case, it suffices to append a transaction T which redeems all the transaction outputs corresponding to $x_1, \ldots, x_n$, and has a *false* script.

Split We implement the *split* action by the script:

```
1  versig(ctxo.owner,rtx.wit)               // check owner
2  and verrec(rtxo(1))                  // covenants on rtx
3  and verrec(rtxo(2))
4  and inlen(rtxo(1)) = 1                      // rtx has 1 in
5  and outlen(rtxo(1)) = 2                   // rtx has 2 outs
6  and rtxo(1).tkval >= 0          // positive token value
7  and rtxo(2).tkval >= 0
8  and rtxo(1).owner = ctxo.owner            // preserve owner
9  and rtxo(1).tkid = ctxo.tkid               // preserve tkid
10 and rtxo(2).tkid = ctxo.tkid
11 and rtxo(1).tkval + rtxo(2).tkval = ctxo.tkval
```

Recall that *split* produces a symbolic step of the form:

$$\langle \mathsf{A}, (v + v') : \tau \rangle_x \xrightarrow{split(x, v, \mathsf{B})} \langle \mathsf{A}, v : \tau \rangle_y \mid \langle \mathsf{B}, v' : \tau \rangle_z$$

Assuming that x corresponds to an unspent transaction output $(\mathsf{T}', 1)$, performing this step in Bitcoin requires to append a transaction T redeeming $(\mathsf{T}', 1)$, and ensuring that:

1. the witness of T carries a signature of the owner (line 1);
2. T has only one input and two outputs, both containing the same script of $(\mathsf{T}', 1)$ (line 2-5);

3. the tkval of T'outputs rtxo(1) and rtxo(2) are ≥ 0 (line 6-7), corresponding to the precondition $v, v' \geq 0$ in [Split];
4. tkid of T's outputs is the same of $(\mathsf{T}', 1)$ (line 9-10);
5. the sum of token values tkval of the outputs of T' is equal to the token value of $(\mathsf{T}', 1)$ (line-11).

Join We implement the *join* action by the script:

```
1 inlen(rtxo(1)) = 2                          // rtx has 2 ins
2 and outlen(rtxo(1)) = 1                     // rtx has 1 out
3 and verrec(rtxo(1))                      // coventant on rtx
4 and verrec(stxo(2))         // covenants on both inputs
5 and verrec(stxo(1))
6 and ctxo.tkid = rtxo(1).tkid        // preserve token id
7 and versig(ctxo.owner, rtx.wit) // check sig of owner
8 and rtxo(1).tkval = stxo(1).tkval + stxo(2).tkval
```

Recall that *join* produces a symbolic step of the form:

$$\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau \rangle_y \xrightarrow{join(x,y,\mathsf{C})} \langle \mathsf{C}, (v + v') : \tau \rangle_z$$

Assume that x and y correspond to the unspent transaction outputs $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$. To perform the corresponding computational step we append a transaction T redeeming $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$, and ensuring that, for both inputs:

1. T has two inputs and one output, containing the same script of $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (line 1-5);
2. the token identifier tkid of the output of T' (rtxo(1)) is the same of $(\mathsf{T}', 1)$ (line 6);
3. the witness of T carries a signature of the owner (line 7);
4. the sum of token values tkval of both inputs is equal to the token value of $(\mathsf{T}, 1)$ (line 8).

Note that the script in $(\mathsf{T}', 1)$ ensures that the one in $(\mathsf{T}'', 1)$ is the same, and vice-versa. In this way, we prevent joining tokens with bitcoins. To also prevent joining tokens of different type, the script checks that the tkid of the current transaction is the same as the one of the first output of the redeeming transaction. This is done by both inputs. In other words, $\mathsf{stxo}(1).\mathsf{tkid} = \mathsf{rtxo}(1).\mathsf{tkid}$ and $\mathsf{stxo}(2).\mathsf{tkid} = \mathsf{rtxo}(1).\mathsf{tkid}$, implying that $\mathsf{stxo}(1).\mathsf{tkid} = \mathsf{stxo}(2).\mathsf{tkid}$.

Exchange We implement the *xchg* action by the script:

```
1  inlen(rtxo(1)) = 2                     // rtx has 2 ins
2  and outlen(rtxo(1)) = 2                // rtx has 2 outs
3  and verrec(stxo(1))               // covenant on input 1
4  and verrec(rtxo(1))                // covenant on rtx(1)
5  and versig(ctxo.owner, rtx.wit)          // check  owner
6  and rtxo(1).owner = stxo(2).owner    // exchange owner
7  and rtxo(2).owner = stxo(1).owner
8  and rtxo(1).tkval = stxo(1).tkval    // preserve value
9  and rtxo(1).tkid = stxo(1).tkid       // preserve tkid
10 if verrec(stxo(2)) then          // exchange token/token
11       verrec(rtxo(2))               // covenant on rtx(2)
12   and rtxo(2).tkval = stxo(2).tkval // preserve tkval
13   and rtxo(2).tkid = stxo(2).tkid    // preserve tkid
14 else                               // exchange token/BTC
15       verscr(versig(ctxo.owner, rtx.wit), rtxo(2))
16   and rtxo(2).val = stxo(2).val        // preserve BTC
```

Recall that the symbolic *xchg* step has the form:

$$\langle \mathsf{A}, v : \tau \rangle_x \mid \langle \mathsf{B}, v' : \tau' \rangle_y \xrightarrow{xchg(x,y)} \langle \mathsf{A}, v' : \tau' \rangle_{x'} \mid \langle \mathsf{B}, v : \tau \rangle_{y'}$$

where τ must be a token, while τ' is either a ₿ or a token.

Assume that x and y correspond to the unspent transaction outputs $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$. To perform the corresponding computational step we append a transaction T redeeming $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$, and ensuring that, for both inputs:

1. T has two inputs and two output (lines 1-2);
2. the first input and the first output of T must contain the same script of $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (lines 3-4);
3. the witness of T carries a signature of the owner (line 5);
4. the owner in the first output of T ($\mathsf{rtxo}(1)$) must be equal to the owner in the second input $(\mathsf{T}'', 1)$ (line 6);
5. dually, the owner in the second output of T ($\mathsf{rtxo}(2)$) must be equal to the owner in the first input $(\mathsf{T}', 1)$ (line 7);
6. the token value and identifier of the first output of T ($\mathsf{rtxo}(1)$) must be equal to those of $(\mathsf{T}', 1)$ (line 8-9).

Furthermore, if $\mathsf{verrec}(\mathsf{stxo}(2))$ is true, i.e. we are exchanging a token with a token. In this case, we require that:

1. the second output of T must contain the same script of $(\mathsf{T}', 1)$ and $(\mathsf{T}'', 1)$ (line 11);
2. the token value and identifier of the second output of T ($\mathsf{rtxo}(2)$) must be equal to those of $(\mathsf{T}'', 1)$ (line 12-13);

When exchanging a token with a ₿ deposit, we require:

1. the script of the second output of T ($\mathsf{rtxo}(2)$) to be $\mathsf{versig}(\mathsf{ctxo}.\mathsf{owner}, \mathsf{rtx}.\mathsf{wit})$ (line 15);
2. the value of the second output of T to be equal to the value of $(\mathsf{T}'', 1)$ (line 16).

Give We implement the *give* action by the script:

```
1 inlen(rtxo(1)) = 1                      // rtx has 1 in
2 and outlen(rtxo(1)) = 1                // rtx has 1 out
3 and versig(ctxo.owner, rtx.wit)         // check  owner
4 and verrec(rtxo(1))               // covenant on rtx(1)
5 and rtxo(1).tkid = ctxo.tkid           // preserve tkid
6 and rtxo(1).tkval = ctxo.tkval        // preserve value
```

Recall that *give* produces a symbolic step of the form:

$$\langle \mathsf{A}, v : \tau \rangle_x \xrightarrow{give(x,\mathsf{B})} \langle \mathsf{B}, v : \tau \rangle_y$$

Assuming that x corresponds to an unspent transaction output $(\mathsf{T}', 1)$, performing this step in Bitcoin requires to append a transaction T redeeming $(\mathsf{T}', 1)$, and ensuring that:

1. T has only one input and one output (lines 1-2);
2. the witness of T carries a signature of the owner (line 3);
3. the output of T ($\mathsf{rtxo}(1)$) contains the same script of $(\mathsf{T}', 1)$ (line 4);
4. the token value and identifier of the first output of T ($\mathsf{rtxo}(1)$) is the same of those of $(\mathsf{T}', 1)$ (lines 5-6).

Efficiency of the implementation To estimate the efficiency of the implementation, we consider the number of cryptographic operations, as their execution cost is an order of magnitude greater than the other operations. In particular, performing verrec and verscr requires to compute the hash of a script (once this is done, the cost of comparing two hashes is negligible). This cost can be reduced by incentivising nodes to cache scripts. The most expensive token action is *xchg*, which, having two inputs, needs to verify 2 signatures and execute at most 10 covenants operations, which overall require to compute at most 6 script hashes. If nodes cache scripts, the cost of the action is not dissimilar to the one required to append a standard transaction with two inputs.

Note that, even though e_{TOK} is a non-standard script, it could be used in a standard P2SH transaction, as in [81], if it did not exceed the

520-bytes limit. Taproot [121] would mitigate this issue: for scripts with multiple disjoint branches, Taproot allows the witness of the redeeming transaction to reveal just the needed branch. Therefore, the 520-bytes limit would apply to branches instead of the whole script.

Conclusions

In this thesis we defined new formal models to improve the security of Bitcoin smart contracts, and developed the BitML toolchain.

In particular, we presented (i) a toolchain for developing BitML contracts (ii) a formal model of Bitcoin which is the foundation for (iii) a new process algebra for defining Bitcoin smart contracts (iv) a new extension to Bitcoin which extends its expressiveness as a smart contract platform, which has been exploited to implement (v) fungible tokens on Bitcoin, complete with a symbolic model.

Main results and future works

A formal model of Bitcoin transactions

We have proposed a formal model for Bitcoin transactions. Our model abstractly describes their essential aspects, at the same time enabling formal reasoning, and providing a formal specification to some of Bitcoin's less documented features.

An alternative model of transactions in blockchain systems has been proposed in [99]. Roughly, blockchains are represented as directed acyclic graphs, where edges denote transfers of assets. This model is quite abstract, so that it can be instantiated to different blockchains (e.g., Bitcoin, Ethereum, and Hyperledger Fabric). Differently from ours, the model in [99] does not capture some peculiar features of Bitcoin, like e.g. transaction signatures and signature modifiers, output scripts, multi-signature verification, and Segregated Witnesses.

Our work provides the theoretical foundations to model Bitcoin smart contracts, reducing the gap between cryptography and programming languages communities. A formal description of smart contracts enables their automated verification and analysis, which are of crucial importance in a context where design flaws may result in loss of money. For instance, our model has been exploited in [9] to present a comprehensive survey of Bitcoin smart contracts.

Differences between our model and Bitcoin There are some differences between our model and the actual Bitcoin, which we outline below.

In Definition 3.3, we stipulate that the in field of a transaction points to another transaction. Instead, in Bitcoin the in field contains the identifier of the input transaction. More specifically, this identifier is defined as $H(\mu(\mathsf{T}))$, where: (i) $\mu = \{\mathsf{wit} \mapsto \bot\}$ since the activation of the SegWit feature; (ii) $\mu = \bot$, beforehand. Consequently, the condition $(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$ item (a) of Definition 3.11 would be translated in Bitcoin as: $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{H}(\mu(\mathsf{T}'')), i)$, where $\mathsf{H}(\mu(\mathsf{T}'')) = \mathsf{H}(\mu(\mathsf{T}))$. Intuitively, the in field specifies the transaction (and the output index) to redeem. Since the activation of SegWit, the computation of the transaction identifier does not take in account the wit field.

The scripting language in Definition 3.1 is a bit more expressive than Bitcoin's. For instance, the script $\lambda x.\mathsf{H}(x) < k$ is admissible in our model, while it is not in Bitcoin. Indeed, the Bitcoin scripting language only admits the comparison (via the OP_LESSTHANOREQUAL opcode) on 32-bit integers, while two arbitrary values can only be tested for equality (via the OP_EQUAL opcode). Similar restrictions apply to arithmetic operations. It is straightforward to adapt our model to apply the same restrictions on Bitcoin scripts. Indeed, our compiler already implements a simple type system which rules away scripts not admissible in Bitcoin.

Definition 3.12 models blockchains as sequences of transactions, while in Bitcoin they are sequences of *blocks* of transactions. In this way, we are abstracting both from the cryptographic puzzle that miners have to solve to append new blocks to the blockchain, and from the *coinbase transactions*, which (like our initial transaction) do not redeem other transactions, and mint new bitcoins (the block rewards). Coinbase transactions are also used in Bitcoin to collect transaction fees, which are just discarded in our model. Extending our model with coinbase transactions would falsify Theorem 3.1.5, since the overall value in the blockchain would no longer be decreasing. Definition 3.12 requires the timestamp of each transaction to increase monotonically. Instead, in Bitcoin a timestamp is valid if it is greater than the median timestamp of previous 11 blocks.

In Definitions 3.3 and 3.11, the absLock and relLock fields specify the time when a transaction can be appended to the blockchain. In Bitcoin transactions, besides the time we can also use the *block height*, i.e. the distance between any given block and the genesis block. Setting the block height to h implies that the transaction can be mined from the block h

onward.

Related works Several works have proposed to use Bitcoin beyond the sole purpose of exchanging currency, by exploiting the flexibility of its scripting language. They propose to implement smart contracts, intended as sets of protocols of the participants involved in them.

Smart contracts requiring external state, namely oracles [87] and escrows [84], can be easily implemented using multi-signature transactions. Such implementations, however, rely on trusted third parties. The work [4] showed that Bitcoin can be used to implement timed commitments through deposit transactions. The commitments are then used to perform multiparty computations [1], such as calculating the winner of a lottery. The main drawback of this approach is indeed the deposit, which grows quadratically with the number of participants. More recently, [62] and [23] have proposed lottery smart contracts that require, respectively, zero and constant (≥ 0) deposit. However, [62] requires the computation of an exponential number of signature w.r.t. the number of participants, while [23] only a quadratic one. The work [12] proposed a contingent payment protocol that can be implemented relying only on standard Bitcoin transaction. It allows to sell solutions for a class of NP problems (e.g. the factorization of a number), the use of zero-knowledge proofs ensure its correctness to the buyer.

Extending Bitcoin with Neighborhood Covenants

We have proposed a formalisation of neighborhood covenants, an extension of traditional Bitcoin covenants in literature. we have exploited our formalisation to present a series of use cases which appear to be unfeasible in pure Bitcoin. We have introduced high-level contract primitives that exploit covenants to enable recursion, and allow contracts to receive new funds and parameters at runtime.

The first proposals of covenants in Bitcoin date back at least to 2013 [114]. Nevertheless, their inclusion into the official Bitcoin protocol is still uncertain, mainly because of the extremely cautious approach to implement changes to Bitcoin [102]. Still, the emerging of Bitcoin layer-2 protocols, like e.g. the Lightning Network [122], has revived the interest in covenants, as witnessed by a recent Bitcoin Improvement Proposal (BIP 119 [123]). The work in [63] propose a new opcode CheckOutputVerify, to explicitly access and constraint the outputs of the redeeming transaction. In contrast, [66] implements covenants exploiting the current implementation of versig, which checks a signature on data that is build by implicitly accessing the redeeming transaction, to define a new operator CheckSigFromStack. Both implementations can require the script of the redeeming transaction to contain the same covenant of the spent one, enable recursive covenants. Another approach is to implement covenants without adding new opcodes [70], which would require a change in the consensus protocol, employing signatures to commit to transaction templates. The drawback of the approach is the need to delete the cryptographic keys used to sign the templates, which adds assumptions to the security model. As noted by [66], recursive covenants would allow to implement Bitcoin contracts that execute state machines, by appending transactions to trigger state transitions. Ergo [35] implement contracts as state machines on a UTXO blockchain using transaction trees. Differently from BitML [10], where transactions of a contract are pre-signed by its users, Ergo can use covenants to enforce the next transaction in the contract execution, enabling loops in the state transition. The work in [18] propose a formal model of covenants, which can be implemented in Bitcoin with modifications similar to the ones in [63, 66]. It exploits the model to specify some complex Bitcoin contracts, and discuss how to exploit covenants to design high-level language primitives for Bitcoin contracts.

This added flexibility can be exploited to design expressive high-level contract languages like Marlowe [68] and Plutus [30].

Known limitations Most of the scripts crafted in our use cases would produce non-standard transactions, that are rejected by Bitcoin nodes. To produce standard transactions from non-standard scripts, we can exploit P2SH [88]. This requires the transaction output to commit to the hash of the script, while the actual script is revealed in the witness of the redeeming transaction. Since, to check its hash, the script needs to be pushed to the stack, and the maximum size of a stack element is 520 bytes, longer scripts would be rejected. This clearly affects the expressiveness of contracts, as already observed in [10]. In particular, since the size of a script grows with the number of contract states (see e.g. Figure 4.10), contracts with many states would easily violate the 520 bytes limit. The introduction of Taproot [121] would mitigate this limit. For scripts with multiple disjoint branches, Taproot allows the witness of the redeeming transaction to reveal just the needed branch. Therefore, the 520 bytes limit would apply to branches, instead of the whole script. Another expressiveness limit derives from the fact that covenants can only constrain the scripts of the redeeming transaction. While this is enough to express non-fungible tokens (see Section 4.2.2), fungible ones seem to require more powerful mechanisms, because of the join operation. An alternative technique to enhancing covenants is to implement fungible tokens natively [33, 32], or to enforce their logic through a sidechain [117].

Other extensions to Bitcoin. Indeed, the Bitcoin scripting language features a very limited set of operations [89], and over the years many useful (and apparently harmless) opcodes have been disabled without a clear understanding of their alleged insecurity [101]. This is the case e.g., of bitwise logic operators, shift operators, integer multiplication, division and modulus. For this reason some developers proposed to re-enable some disabled opcodes [78], and some works in the literature proposed extensions to the Bitcoin scripting language so to enhance the expressiveness of smart contracts.

Secure cash distribution with penalties [57, 4, 26] is a cryptographic primitive which allows a set of participants to make a deposit, and then provide inputs to a function whose evaluation determines how the deposits are distributed among the participants. This primitive guarantees that dishonest participants (who, e.g., abort the protocol after learning the value of the function) will pay a penalty to the honest participants. This primitive does not seem to be directly implementable in Bitcoin, but it becomes so by extending the scripting language with the opcode CHECKSIGFROMSTACK discussed above. Secure cash distribution with penalties can be instantiated to a variety of smart contracts, e.g. lotteries [4] poker [57], and contingent payments. The latter smart contract can also be obtained through the opcode CHECKKEYPAIRVERIFY in [38], which checks if the two top elements of the stack are a valid key pair.

Another new opcode, called MULTIINPUT [62] consumes from the stack a signature σ and a sequence of in values $(\mathsf{T}_1, j_1) \cdots (\mathsf{T}_n, j_n)$, with the following two effects: (i) it verifies the signature σ against the redeeming transaction T, neglecting $\mathsf{T}.\mathsf{in}$; (ii) it requires $\mathsf{T}.\mathsf{in}$ to be equal to some of the T_i. Exploiting this opcode, [62] devise a fair N-party lottery which requires zero deposit, and $O(N^2)$ off-chain signed transaction. The first one of these effects can be alternatively obtained by extending, instead of the scripting language, the signature modifiers. More specifically, [23] introduces a new signature modifier, which can set to $\perp$ *all* the inputs of a transaction (i.e., no input is signed). In this way they obtain a fair multi-party lottery with similar properties to the one in [62].

Another way to improve the expressiveness of smart contracts is to replace the Bitcoin scripting language, e.g. with the one in [65]. This would also allow to establish bounds on the computational resources needed to run scripts.

Unfortunately, none of the proposed extensions has been yet included in the main branch of the Bitcoin Core client, and nothing suggests that they will be considered in the near future. Indeed, the development of Bitcoin is extremely conservative, as any change to its protocol requires an overwhelming consensus of the miners. So far, new opcodes can only be empirically assessed through the Elements alpha project[1], a testnet for experimenting new Bitcoin features. A significant research challenge would be that of formally proving that new opcodes do not introduce vulnerabilities, exploitable e.g. by Denial-of-Service attacks. For instance, unconstrained uses of the opcode `CAT` may cause an exponential space blow-up in the verification of transactions.

Verification Although designing contracts in the UTXO model seems to be less error-prone than in the shared memory model, e.g. because of the absence of reentrancy vulnerabilities (like the one exploited in the Ethereum DAO attack [124]), Bitcoin contracts may still contain security flaws. Therefore, it is important to devise verification techniques to detect security issues that may lead to the theft or freezing of funds. Recursive covenants make this task harder than in pure Bitcoin, since they can encode infinite-state transition systems, as in most of our use cases. Hence, model-checking techniques based on the exploration of the whole state space, like the one used in [3], cannot be applied.

High-level Bitcoin contracts The compiler of our extension of BitML is just sketched in Section 4.3, and we leave as future work its formal definition, as well as the extension of the computational soundness results of [22], ensuring the correspondence between the symbolic semantics of BitML and the underlying computational level of Bitcoin. Continuing along this line of research, it would be interesting to study new linguistic primitives that fully exploit the expressiveness of Bitcoin covenants, and to extend accordingly the verification technique of [24]. Note that our extension of the UTXO model is more restrictive than the one in [31], as the latter abstracts from the script language, just assuming that scripts denote any pure functions [126]. This added flexibility can be exploited to design expressive high-level contract languages like Marlowe [68] and Plutus [30].

Bitcoin smart contracts as endpoint protocols

The formal model of smart contracts we have proposed in Chapter 5 is based on the current mechanisms of Bitcoin; indeed, this makes it possible to translate endpoint protocols into actual implementations interacting with the Bitcoin blockchain. However, constraining smart contracts to perfectly adhere to Bitcoin greatly reduces their expressiveness.

As witnessed in Section 5.2, designing secure smart contracts on Bitcoin is an error-prone task, similarly to designing secure cryptographic protocols. The reason lies in the fact that, to devise a secure contract, a designer has to anticipate any possible (mis-)behaviour of the other participants. The side effect is that endpoint protocols may be quite convoluted, as they must include compensations at all the points where something can go wrong. Therefore, tools

[1] `https://elementsproject.org/elements/opcodes/`

to automate the analysis and verification of smart contracts may be of great help.

Recent works [2] propose to verify Bitcoin smart contracts by modelling the behaviour of participants as timed automata, and then using UPPAAL [25] to check properties against an attacker. This approach correctly captures the time constraints within the contracts. The downside is that encoding this UPPAAL model into an actual implementation with Bitcoin transactions is a complex task. Indeed, a designer without a deep knowledge of Bitcoin technicalities is likely to produce an UPPAAL model that can *not* be encoded in Bitcoin. A relevant research challenge is to study specification languages for Bitcoin contracts (like e.g. the one in Section 5.1), and techniques to *automatically* encode them in a model that can be verified by a model checker.

Remarkably, the verification of security properties of smart contracts requires to deal with non-trivial aspects, like temporal constraints and probabilities. This is the case, e.g., for the verification of fairness of lotteries (like e.g. the one discussed in Section 5.2.8); a further problem is that fairness must hold against any adversarial strategy. It is not clear whether in this case it is sufficient to consider a "most powerful" adversary, like e.g. in the symbolic Dolev-Yao model. In case a contract is not secure against arbitrary (PTIME) adversaries, one would like to verify that, at least, it is secure against *rational* ones [45], which is a relevant research issue. Additional issues arise when considering more concrete models of the Bitcoin blockchain, respect to the one in Section 1.1. This would require to model *forks*, i.e. the possibility that a recent transaction is removed from the blockchain. This could happen with rational (but dishonest) miners [59].

DSLs for smart contracts. As witnessed in Section 5.2, modelling Bitcoin smart contracts is complex and error-prone. A possible way to address this complexity is to devise high-level domain-specific languages (DSLs) for contracts, to be compiled in low-level protocols (e.g., the ones in Section 5.1). Indeed, the recent proliferation of non-Turing complete DSLs for smart contracts [97, 43, 28] suggests that this is an emerging research direction.

A first proposal of an high-level language implemented on top of Bitcoin is Typecoin [37]. This language allows to model the updates of a state machine as affine logic propositions. Users can "run" this machine by putting transactions on the Bitcoin blockchain. The security of the blockchain guarantees that only the legit updates of the machine can be triggered by users. A downside of this approach is that liveness is guaranteed only by assuming cooperation among the participants, i.e., a dishonest participant can make the others unable to complete an execution. Note instead that the smart contracts in Section 5.2 allow honest participants to terminate, regardless of the behaviours of the environment. In some cases, e.g. in the lottery in Section 5.2.8, abandoning the contract may even result in penalties (i.e., loss of the deposit paid upfront to stipulate the contract).

A toolchain for BitML

While the business of smart contracts has flourished on platforms like Ethereum and Cardano, it never caught on Bitcoin. One of the main reasons is that, unlike the other platforms, Bitcoin has neither high-level contract languages, nor related development and verification tools.

A downside of using platforms with expressive, Turing-complete languages, is that they may expose contracts to a wider attack surface: indeed, a series of language-induced vulnerabilities of Ethereum contracts [8] has caused losses of hundreds of millions of USD [124, 120, 77].

Although our benchmarks witness a rich variety of contracts expressible in BitML, there is room for improvement. BitML is not *Bitcoin-complete*, i.e. some contracts executable in Bitcoin are not expressible in BitML. The main sources of this incompleteness are three: (i) all the transactions obtained by the compiler must be signed *before* stipulation by all the involved participants (only the signatures for authorizations can be provided at run-time); (ii) all transaction fields must be taken into account when computing signatures, while partial signatures (e.g. those obtained through SIGHASH_ANYONECANPAY and SIGHASH_SINGLE) are not used; (iii) off-chain interactions are limited to revealing secrets and providing authorizations. The first constraint is required to ensure that honest participants can always perform, at the Bitcoin level, the moves enabled in the corresponding BitML contract, regardless of the behaviour of the others. In this respect, BitML follows the standard assumption that participants are *non-cooperative*, i.e. at any moment after stipulation they can stop interacting (unlike TypeCoin [37], which assumes cooperation, allowing dishonest participants to make a contract deadlock). Yet, cooperation can be incentivized, by punishing misbehaviour with penalties, like e.g. in the timed commitment of Section 6.1. As a consequence of the design choices above, contracts with a dynamically-defined set of players (e.g., crowdfunding), or an unbounded number of iterations (e.g., micro-payment channels), are not expressible in BitML.

The limitations of BitML (and of Bitcoin) could be overcome in various ways. For instance, using Bitcoin "as-is", it would be possible to relax constraint (iii) above, so to allow e.g. zero-knowledge off-chain protocols. This would enable to extend BitML with primitives to express *contingent payments* contracts, where participants trade solutions of a class of NP problems [12, 115]. Similarly, by relaxing constraint (i), we could extend BitML to enable dynamic stipulation of subcontracts, requiring that all the involved participants provide their signatures at run-time. This would allow to model e.g. micro-payment channels in BitML. Together with the use of SIGHASH_ANYONECANPAY (relaxing constraint (ii)), this would also allow for modelling crowdfunding contracts. As before, this extension could be implemented without modifying Bitcoin.

Other extensions of BitML would require extensions of Bitcoin. For instance, *covenants* [63, 66] would allow for implementing arbitrary finite-state machines. Controlled *input malleability* would allow to efficiently implement tournaments in multi-player gambling games, like e.g. lotteries [23]. This can also be achieved through a new opcode that checks if the redeeming transaction belongs to a given set [62]. Contingent payments without zero-knowledge proofs can be achieved by exploiting a new opcode that checks the validity of key pairs [38]. A new opcode which checks signatures for arbitrary messages would allow for expressing general fair multiparty computations [57]. Further, fair and robust multiparty computations can be achieved using more complex transactions [54]. A more radical approach would be to replace the Bitcoin scripting language with a more expressive one, like e.g. Simplicity [65].

Compared with the tools for analysing Ethereum contracts [60, 73, 48, 50, 67, 49, 27, 69], whose precision is subject to the limitations derived by the Turing-completeness of the underlying languages, our toolchain features a sound and complete verification technique.

On-chain Fungible Tokens on Bitcoin

We have proposed a secure and efficient implementation of fungible tokens on Bitcoin, exploiting neighbourhood covenants, a powerful yet simple extension of the Bitcoin script language. We formalise fungible tokens, including a symbolic model for fungible token, which can be applied to UTXO-based blockchains, and we prove some correctness properties of the model.

To keep the presentation simple, we have limited the functionality of tokens a bit, making split/join/exchange actions operate on just two deposits, and omitting time constraints. Removing these restrictions would only affect the size, but not the complexity, of our technical development. Further, it would allow to use tokens *as is* within high-level languages for Bitcoin contracts, e.g. BitML [22], simplifying the design of financial contracts which manage tokens and bitcoins. For instance, we would express as follows a basic zero-coupon bond [53] where an investor A pays upfront to a bank B 5 units of token τ, and receives back 1₿ after a maturity date t:

$$\texttt{split}\ \big(5\tau \rightarrow \texttt{withdraw}\ \mathsf{B} \ \mid\ 2₿ \rightarrow \texttt{after}\, t : \texttt{withdraw}\ \mathsf{A}\big)$$

A research question arising from our work is how to exploit neighbourhood covenants in the design of high-level languages for Bitcoin contracts. Besides enhancing the expressiveness of these languages, neighbourhood covenants would enable a simpler compilation technique, compared e.g. that used by BitML, reducing the off-chain exchange of signatures.

Related work In Ethereum, similarly to our approach, tokens are implemented on top of the platform using custom code, following the ERC-20 and ERC-712 standards [106, 104]. Instead, the works in [33, 32] propose a generalisation of the EUTXO model [31] that natively supports custom tokens. Similarly, the work in [127] proposes an UTXO model that natively support tokens in the same way it supports the main cryptocurrency. In the model, each token or currency have the same status: fees for a token transaction are paid directly with some units of the token itself, making the currencies mutually independent.

List of Figures

List of Tables

Bibliography

References

[1] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. “Fair Two-Party Computations via Bitcoin Deposits”. In: *Financial Cryptography Workshops*. Vol. 8438. LNCS. Springer, 2014, pp. 105–121. DOI: `10.1007/978-3-662-44774-1\_8`.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. “Modeling bitcoin contracts by timed automata”. In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2014, pp. 7–22.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. “Modeling Bitcoin contracts by timed automata”. In: *FORMATS*. Vol. 8711. LNCS. Springer, 2014, pp. 7–22. DOI: `10.1007/978-3-319-10512-3\_2`.

[4] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. “Secure Multiparty Computations on Bitcoin”. In: *IEEE S & P*. 2014, pp. 443–458. DOI: `10.1109/SP.2014.35`.

[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. “Secure multiparty computations on Bitcoin”. In: *Commun. ACM* 59.4 (2016), pp. 76–84. DOI: `10.1145/2896386`.

[6] Monika Di Angelo and Gernot Salzer. “Tokens, Types, and Standards: Identification and Utilization in Ethereum”. In: *DAPPS*. IEEE, 2020, pp. 1–10. DOI: `10.1109/DAPPS49028.2020.00001`.

[7] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts SoK". In: *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. ISBN: 978-3-662-54454-9. DOI: 10.1007/978-3-662-54455-6_8. URL: https://doi.org/10.1007/978-3-662-54455-6_8.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)". In: *Principles of Security and Trust (POST)*. Vol. 10204. LNCS. Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8.

[9] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. "SoK: unraveling Bitcoin smart contracts". In: *POST*. Vol. 10804. LNCS. Springer, 2018, pp. 217–242. DOI: 10.1007/978-3-319-89722-6.

[10] Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. "Developing secure Bitcoin contracts with BitML". In: *ESEC/FSE*. 2019. DOI: https://doi.org/10.1145/3338906.3341173.

[11] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "A formal model of Bitcoin transactions". In: *Financial Cryptography and Data Security*. Vol. 10957. LNCS. Springer, 2018. DOI: 10.1007/978-3-662-58387-6.

[12] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. "Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts". In: *ESORICS*. Vol. 9879. LNCS. Springer, 2016, pp. 261–280. DOI: 10.1007/978-3-319-45741-3_14.

[13] Massimo Bartoletti, Bryn Bellomy, and Livio Pompianu. "A Journey into Bitcoin Metadata". In: *J. Grid Comput.* 17.1 (2019), pp. 3–22. DOI: 10.1007/s10723-019-09473-3.

[14] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. "Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact". In: *Future Gener. Comput. Syst.* 102 (2020), pp. 259–277. DOI: 10.1016/j.future.2019.08.014.

[15] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. "Fun with Bitcoin Smart Contracts". In: *ISoLA*. 2018, pp. 432–449. DOI: 10.1007/978-3-030-03427-6_32.

[16] Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. "Verification of recursive Bitcoin contracts". In: *Submitted to LMCS* (2020). arXiv: 2011.14165.

[17] Massimo Bartoletti, Stefano Lande, and Alessandro Sebastian Podda. "A Proof-of-Stake protocol for consensus on Bitcoin subchains". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 568–584.

[18] Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "Bitcoin Covenants Unchained". In: *ISoLA*. To appear. 2020.

[19] Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "Computationally sound Bitcoin tokens". In: *IEEE Computer Security Foundations Symposium*. To appear. 2021. arXiv: 2010.01347.

[20] Massimo Bartoletti, Maurizio Murgia, and Roberto Zunino. "Renegotiation and recursion in Bitcoin contracts". In: *COORDINATION*. Vol. 12134. LNCS. Springer, 2020, pp. 261–278. DOI: 10.1007/978-3-030-50029-0_17.

[21] Massimo Bartoletti and Livio Pompianu. "An Analysis of Bitcoin OP_RETURN Metadata". In: *Financial Cryptography and Data Security*. Ed. by Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson. Cham: Springer International Publishing, 2017, pp. 218–230. ISBN: 978-3-319-70278-0.

[22] Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *ACM CCS*. 2018. DOI: 10.1145/3243734.3243795.

[23] Massimo Bartoletti and Roberto Zunino. "Constant-deposit multiparty lotteries on Bitcoin". In: *Financial Cryptography Workshops*. Vol. 10323. LNCS. Springer, 2017. DOI: 10.1007/978-3-319-70278-0.

[24] Massimo Bartoletti and Roberto Zunino. "Verifying liquidity of Bitcoin contracts". In: *POST*. Vol. 11426. LNCS. Springer, 2019, pp. 222–247.

[25] Gerd Behrmann, Alexandre David, and Kim G Larsen. "A tutorial on Uppaal". In: *Formal methods for the design of real-time systems*. Vol. 3185. LNCS. Springer, 2004, pp. 200–236. DOI: 10.1007/978-3-540-30080-9_7.

[26] Iddo Bentov and Ranjit Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *CRYPTO*. Vol. 8617. LNCS. Springer, 2014, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[27] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Beguelin. "Formal Verification of Smart Contracts". In: *PLAS*. 2016.

[28] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. "Findel: Secure Derivative Contracts for Ethereum". In: *Financial Cryptography Workshops*. Vol. 10323. LNCS. Springer, 2017, pp. 453–467. DOI: 10.1007/978-3-319-70278-0_28.

[29] Dan Boneh and Moni Naor. "Timed Commitments". In: *CRYPTO*. Vol. 1880. LNCS. Springer, 2000, pp. 236–254. DOI: 10.1007/3-540-44598-6.

[30] Lars Brünjes and Murdoch James Gabbay. "UTxO- vs account-based smart contract blockchain programming paradigms". In: *CoRR* abs/2003.14271 (2020). arXiv: 2003.14271.

[31] Manuel M.T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. "The Extended UTXO Model". In: *Financial Cryptography Workshops*. To appear. 2020.

[32] Manuel M.T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. "Native Custom Tokens in the Extended UTXO Model". In: *ISoLA*. To appear. 2020.

[33] 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*. To appear. 2020.

[34] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology". In: *WWW*. ACM, 2018, pp. 1409–1418. DOI: 10.1145/3178876.3186046.

[35] Alexander Chepurnoy and Amitabh Saxena. "Multi-stage Contracts in the UTXO Model". In: *Cryptocurrencies and Blockchain Technology*. Vol. 11737. LNCS. Springer, 2019, pp. 244–254. DOI: 10.1007/978-3-030-31500-9_16.

[36] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. “Maude: Specification and Programming in Rewriting Logic”. In: *TCS* (2001).

[37] Karl Crary and Michael J. Sullivan. “Peer-to-peer affine commitment using Bitcoin”. In: *ACM Conf. on Programming Language Design and Implementation*. 2015, pp. 479–488. DOI: 10.1145/2737924.2737997.

[38] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. “A fair protocol for data trading based on Bitcoin transactions”. In: *Future Generation Computer Systems* (2017). DOI: 10.1016/j.future.2017.08.021.

[39] Kevin Delmolino, Mitchell Arnett, Andrew Millerand Ahmed Kosba, and Elaine Shi. “Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab”. In: (2016).

[40] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. “The Maude LTL Model Checker”. In: *Electr. Notes Theor. Comput. Sci.* 71 (2002), pp. 162–187. DOI: 10.1016/S1571-0661(05)82534-4.

[41] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. “Counter machines and counter languages”. In: *Mathematical systems theory* 2.3 (1968), pp. 265–283. DOI: 10.1007/BF01694011.

[42] Matthew Flatt. “Creating languages in Racket”. In: *Commun. ACM* 55.1 (2012), pp. 48–56. DOI: 10.1145/2063176.2063195.

[43] C. K. Frantz and M. Nowostawski. “From Institutions to Code: towards Automated Generation of Smart Contracts”. In: *eCAS Workshop*. 2016.

[44] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. “Detecting Token Systems on Ethereum”. In: *Financial Cryptography and Data Security*. Vol. 11598. LNCS. Springer, 2019, pp. 93–112. DOI: 10.1007/978-3-030-32101-7_7.

[45] Juan A. Garay, Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. “Rational Protocol Design: Cryptography against Incentive-Driven Adversaries”. In: *FOCS*. 2013, pp. 648–657. DOI: 10.1109/FOCS.2013.75.

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *ACM Symposium on Theory of Computing.* ACM, 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[47] David M. Goldschlag, Stuart G. Stubblebine, and Paul F. Syverson. "Temporarily hidden bit commitment and lottery applications". In: *Int. J. Inf. Sec.* 9.1 (2010), pp. 33–50. DOI: 10.1007/s10207-009-0094-1.

[48] 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).* Vol. 10804. LNCS. Springer, 2018, pp. 243–269. DOI: 10.1007/978-3-319-89722-6_10.

[49] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "Foundations and Tools for the Static Analysis of Ethereum Smart Contracts". In: *CAV.* Vol. 10981. LNCS. Springer, 2018, pp. 51–78. DOI: 10.1007/978-3-319-96145-3_4.

[50] Everett Hildenbrandt et al. "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine". In: *IEEE Computer Security Foundations Symposium (CSF).* IEEE Computer Society, 2018, pp. 204–217. DOI: 10.1109/CSF.2018.00022.

[51] Everett Hildenbrandt et al. "KEVM: A Complete Semantics of the Ethereum Virtual Machine". In: *2018 IEEE 31st Computer Security Foundations Symposium.* IEEE, 2018, pp. 204–217.

[52] Yoichi Hirai. "Defining the Ethereum Virtual Machine for Interactive Theorem Provers". In: *Financial Cryptography Workshops.* Vol. 10323. LNCS. Springer, 2017, pp. 520–535. DOI: 10.1007/978-3-319-70278-0_33.

[53] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. "Composing contracts: an adventure in financial engineering, functional pearl". In: *International Conference on Functional Programming (ICFP).* 2000, pp. 280–292. DOI: 10.1145/351240.351267.

[54] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. "Fair and Robust Multi-party Computation Using a Global Transaction Ledger". In: *EUROCRYPT.* Vol. 9666. LNCS. Springer, 2016, pp. 705–734. DOI: 10.1007/978-3-662-49896-5_25.

[55] Ranjit Kumaresan and Iddo Bentov. "Amortizing Secure Computation with Penalties". In: *ACM CCS.* 2016, pp. 418–429. DOI: 10.1145/2976749.2978424.

[56] Ranjit Kumaresan and Iddo Bentov. "How to Use Bitcoin to Incentivize Correct Computations". In: *ACM CCS*. 2014, pp. 30–41. DOI: 10.1145/2660267.2660380.

[57] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. "How to Use Bitcoin to Play Decentralized Poker". In: *ACM CCS*. 2015, pp. 195–206. DOI: 10.1145/2810103.2813712.

[58] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. "Improvements to Secure Computation with Penalties". In: *ACM CCS*. 2016, pp. 406–417. DOI: 10.1145/2976749.2978421.

[59] Kevin Liao and Jonathan Katz. "Incentivizing Blockchain Forks via Whale Transactions". In: *Financial Cryptography Workshops*. Vol. 10323. LNCS. Springer, 2017, pp. 264–279. DOI: 10.1007/978-3-319-70278-0_17.

[60] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making Smart Contracts Smarter". In: *ACM CCS*. 2016, pp. 254–269. DOI: 10.1145/2976749.2978309.

[61] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. "A fistful of Bitcoins: characterizing payments among men with no names". In: *Commun. ACM* 59.4 (2016), pp. 86–93. DOI: 10.1145/2896384.

[62] Andrew Miller and Iddo Bentov. "Zero-Collateral Lotteries in Bitcoin and Ethereum". In: *EuroS&P Workshops*. 2017, pp. 4–13. DOI: 10.1109/EuroSPW.2017.44.

[63] Malte Möser, Ittay Eyal, and Emin Gün Sirer. "Bitcoin covenants". In: *Financial Cryptography Workshops*. Vol. 9604. LNCS. Springer, 2016, pp. 126–141. DOI: 10.1007/978-3-662-53357-4_9.

[64] Xavier Nicollin and Joseph Sifakis. "An Overview and Synthesis on Timed Process Algebras". In: *CAV*. Vol. 575. LNCS. 1991, pp. 376–398. DOI: 10.1007/3-540-55179-4_36.

[65] Russell O'Connor. "Simplicity: A New Language for Blockchains". In: *PLAS*. 2017. URL: http://arxiv.org/abs/1711.03028.

[66] Russell O'Connor and Marta Piekarska. "Enhancing Bitcoin transactions with covenants". In: *Financial Cryptography Workshops*. Vol. 10323. LNCS. Springer, 2017. DOI: 10.1007/978-3-319-70278-0_12.

[67] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Rosu. "A formal verification tool for Ethereum VM bytecode". In: *ACM ESEC/SIGSOFT FSE*. 2018, pp. 912–915. DOI: 10.1145/3236024.3264591.

[68] Pablo Lamela Seijas and Simon J. Thompson. "Marlowe: Financial Contracts on Blockchain". In: *ISoLA*. Vol. 11247. LNCS. Springer, 2018, pp. 356–375. DOI: 10.1007/978-3-030-03427-6_27.

[69] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a Smart Contract Intermediate-Level LAnguage". In: *CoRR* abs/1801.00687 (2018).

[70] Jacob Swambo, Spencer Hommel, Bob McElrath, and Bryan Bishop. "Bitcoin Covenants: Three Ways to Control the Future". In: *CoRR* abs/2006.16714 (2020).

[71] Paul F. Syverson. "Weakly Secret Bit Commitment: Applications to Lotteries and Fair Exchange". In: *IEEE CSFW*. 1998, pp. 2–13. DOI: 10.1109/CSFW.1998.683149.

[72] Nick Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 2.9 (1997). http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548.

[73] Petar Tsankov, Andrei Marian Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. "Securify: Practical Security Analysis of Smart Contracts". In: *CoRR* abs/1806.01143 (2018).

[74] Marie Vasek and Tyler Moore. "There's No Free Lunch, Even Using Bitcoin: Tracking the Popularity and Profits of Virtual Currency Scams". In: *Financial Cryptography and Data Security*. 2015, pp. 44–61. DOI: 10.1007/978-3-662-47854-7_4.

[75] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

Online references

[76] *520-byte limitation on serialized script size*. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.

mediawiki#520-byte-limitation-on-serialized-script-size.

[77] *A Postmortem on the Parity Multi-Sig Library Self-Destruct.* https://goo.gl/Kw3gXi. 11/2017.

[78] *A proposal to reintroduce the disabled script opcodes.* URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014356.html.

[79] *Algorand Developer Docs - Assets.* https://developer.algorand.org/docs/features/asa/. 2020.

[80] Adam Back and Iddo Bentov. *Note on fair coin toss via Bitcoin.* http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf. 2013.

[81] *BALZaC: Bitcoin Abstract Language, analyZer and Compiler.* https://blockchain.unica.it/balzac/. 2018.

[82] *Bitcoin Avgerage Transaction Fee historical chart.* URL: https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

[83] *Bitcoin Core integration/staging tree.* URL: https://github.com/bitcoin/bitcoin (visited on 09/25/2018).

[84] *Bitcoin developer guide - Escrow and arbitration.* https://bitcoin.org/en/developer-guide#escrow-and-arbitration. 2015.

[85] *Bitcoin standard transactions.* URL: https://developer.bitcoin.org/devguide/transactions.html#standard-transactions.

[86] *Bitcoin wiki - Contracts - Assurance contracts.* https://en.bitcoin.it/wiki/Contract#Example_3:_Assurance_contracts. 2012.

[87] *Bitcoin wiki - Contracts - Using external state.* https://en.bitcoin.it/wiki/Contract#Example_4:_Using_external_state. 2012.

[88] *Bitcoin wiki - Pay-to-Script Hash.* https://en.bitcoinwiki.org/wiki/Pay-to-Script_Hash. 2017.

[89] *Bitcoin wiki script.* URL: https://en.bitcoin.it/wiki/Script (visited on 10/01/2018).

[90] BitFury group. *Smart Contracts on Bitcoin Blockchain.* http://bitfury.com/content/5-white-papers-research/contracts-1.1.1.pdf. 2015.

[91] *BitML benchmarks contrats.* URL: https://github.com/bitml-lang/bitml-compiler/tree/master/examples/benchmarks.

[92] *BitML compiler.* URL: https://github.com/bitml-lang/bitml-compiler/.

[93] *BitML examples.* URL: https://github.com/bitml-lang/bitml-compiler/tree/master/examples.

[94] *BitML Online Documentation.* URL: https://blockchain.unica.it/bitml/docs/index.html.

[95] *BitML toolchain repository.* URL: https://github.com/bitml-lang/.

[96] *BitML verifier.* URL: https://github.com/bitml-lang/bitml-maude.

[97] Richard G. Brown, James Carlyle, Ian Grigg, and Mike Hearn. *Corda: An Introduction.* http://r3cev.com/s/corda-introductory-whitepaper-final.pdf. 2016.

[98] Vitalik Buterin. *Ethereum: a next generation smart contract and decentralized application platform.* https://github.com/ethereum/wiki/wiki/White-Paper. 2013.

[99] Christian Cachin, Angelo De Caro, Pedro Moreno-Sanchez, Björn Tackmann, and Marko Vukolić. *The Transaction Graph for Modeling Blockchain Semantics.* Cryptology ePrint Archive, Report 2017/1070. 2017. URL: https://eprint.iacr.org/2017/1070.

[100] *Colu website.* https://www.colu.com/. 2020.

[101] *CVE-2010-5141.* URL: https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures\#CVE-2010-5141.

[102] Luke Dashjr. *BIP 0002.* https://en.bitcoin.it/wiki/BIP_0002. 2016.

[103] Robby Dermody, Adam Krellenstein, Ouziel Slama, and Evan Wagner. *CounterParty: Protocol Specification.* 2014. URL: http://counterparty.io/docs/protocol\%5Fspecification/.

[104] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. *EIP 721: ERC-721 Non-Fungible Token Standard.* https://eips.ethereum.org/EIPS/eip-721.

[105] *EPOBC protocol specification.* https://github.com/chromaway/ngcccbase/wiki/EPOBC_simple. 2020.

[106] *ERC-20 Token Standard.* https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. 2015.

[107] *Ethereum token dynamics.* https://stat.bloxy.info/superset/dashboard/tokens.

[108] Mike Hearn. *Rapidly-adjusted (micro)payments to a pre-determined party.* 2013. URL: bitcointalk.org.

[109] *Hyperledger - Open Source Blockchain Technologies.* URL: https://www.hyperledger.org/ (visited on 09/25/2018).

[110] Rosco Kalis. *CashScript — Writing covenants.* 2019. URL: https://cashscript.org/docs/guides/covenants/.

[111] *King of the Ether Throne: Post mortem investigation.* URL: https://www.kingoftheether.com/postmortem.html.

[112] Johnson Lau. *Upgrading Bitcoin: Segregated Witness.* URL: https://www.bitcoinhk.org/media/presentations/2016-03-16/2016-03-16-Segregated\%5FWitness.pdf (visited on 09/25/2018).

[113] Eric Lombrozo, Johnson Lau, and Pieter Wuille. *Segregated Witness (Consensus layer).* BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki. 2015.

[114] Gregory Maxwell. *CoinCovenants using SCIP signatures, an amusingly bad idea.* https://bitcointalk.org/index.php?topic=278122.0. 2013.

[115] Gregory Maxwell. *The first successful Zero-Knowledge Contingent Payment.* https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/. 2016.

[116] Satoshi Nakamoto. *Bitcoin: a peer-to-peer electronic cash system.* https://bitcoin.org/bitcoin.pdf. 2008.

[117] Jonas Nick, Andrew Poelstra, and Gregory Sanders. *Liquid: a Bitcoin Sidechain.* https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf. 2020.

[118] *OpenAssets protocol.* https://github.com/OpenAssets. 2020.

[119] *Oracolize website.* URL: https://www.oraclize.it.

[120] *Parity Wallet Security Alert.* https://paritytech.io/blog/security-alert.html. 07/2017.

[121] Anthony Towns Pieter Wuille Jonas Nick. *Taproot: SegWit version 1 spending rules.* BIP 341, https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki. 2020.

[122] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. 2015. URL: https://lightning.network/lightning-network-paper.pdf.

[123] Jeremy Rubin. *CHECKTEMPLATEVERIFY*. BIP 119, https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki. 2020.

[124] *Understanding the DAO attack*. http://www.coindesk.com/understanding-dao-hack-journalists/. 06/2016.

[125] Gavin Wood. *Ethereum: a secure decentralised generalised transaction ledger*. gavwood.com/paper.pdf. 2014.

[126] Joachim Zahnentferner. *An Abstract Model of UTxO-based Cryptocurrencies with Scripts*. 2018. URL: https://eprint.iacr.org/2018/469.

[127] Joachim Zahnentferner. *Multi-Currency Ledgers*. Cryptology ePrint Archive, Report 2020/895. https://eprint.iacr.org/2020/895. 2020.

Appendices

7.4 Supplementary material for Chapter 7

Full token script The full token script is the following:

```
if not verrec(ptxo(1)) then                  // ctxo is a token generator
      ctxo.op = 0                                          // gen action
  and ctxo.tkid = txid(ptxo(1))                              // token id
  and ptxo(1).val = 0                                // spent txo has 0 BTC
  and outlen(ctxo) = 1                                   // gen has 1 out
  and ctxo.tkval > 0                                // positive token value
else
if rtxo.op = 1 then                /*************** BURN **************/
      versig(ctxo.owner,rtx.wit)                           // check owner
  and verscr(false,rtxo(1))                        // make rtx unspendable
  and inlen(rtxo(1)) = 1                                  // rtx has 1 in
  and outlen(rtxo(1)) = 1                                // rtx has 1 out
else if rtxo.op = 2 then          /************** SPLIT ***************/
      versig(ctxo.owner,rtx.wit)                           // check owner
  and verrec(rtxo(1))                                  // covenants on rtx
  and verrec(rtxo(2))
  and inlen(rtxo(1)) = 1                                  // rtx has 1 in
  and outlen(rtxo(1)) = 2                               // rtx has 2 outs
  and rtxo(1).tkval >= 0                            // positive token value
  and rtxo(2).tkval >= 0
  and rtxo(1).owner = ctxo.owner                        // preserve owner
  and rtxo(1).tkid = ctxo.tkid                           // preserve tkid
  and rtxo(2).tkid = ctxo.tkid
  and rtxo(1).tkval + rtxo(2).tkval = ctxo.tkval        // preserve value
else if rtxo.op = 3 then           /************** JOIN ***************/
      inlen(rtxo(1)) = 2                                  // rtx has 2 in
  and outlen(rtxo(1)) = 1                                // rtx has 1 out
  and verrec(rtxo(1))                                   // coventant on rtx
  and verrec(stxo(2))                           // covenants on both inputs
  and verrec(stxo(1))
  and ctxo.tkid = rtxo(1).tkid                      // preserve token id
  and versig(ctxo.owner, rtx.wit)                         // check owner
  and rtxo(1).tkval = stxo(1).tkval + stxo(2).tkval //preserve value
else if rtxo.op = 4 then           /************** XCHG ***************/
      inlen(rtxo(1)) = 2                                 // rtx has 2 ins
  and outlen(rtxo(1)) = 2                               // rtx has 2 outs
  and versig(ctxo.owner, rtx.wit)                         // check owner
  and verrec(stxo(1))                                 // covenant on input 1
  and verrec(rtxo(1))                                // covenant on rtx(1)
  and rtxo(1).owner = stxo(2).owner                          // swap owners
  and rtxo(2).owner = stxo(1).owner
  and rtxo(1).tkval = stxo(1).tkval                     // preserve value
  and rtxo(1).tkid = stxo(1).tkid                        // preserve tkid
  if verrec(stxo(2)) then         /***** EXCHANGE TOKEN/TOKEN *****/
        verrec(rtxo(2))                              // covenant on rtx(2)
    and rtxo(2).tkval = stxo(2).tkval                   // preserve tkval
    and rtxo(2).tkid = stxo(2).tkid                      // preserve tkid
  else                                  /***** EXCHANGE TOKEN/BTC *****/
        verscr(versig(ctxo.owner, rtx.wit), rtxo(2))
    and rtxo(2).val = stxo(2).val                          // preserve BTC
else if rtxo.op = 5 then           /************** GIVE ***************/
      inlen(rtxo(1)) = 1                                  // rtx has 1 in
  and outlen(rtxo(1)) = 1                                // rtx has 1 out
  and versig(ctxo.owner, rtx.wit)                         // check owner
  and verrec(rtxo(1))                                // covenant on rtx(1)
  and rtxo(1).tkid = ctxo.tkid                           // preserve tkid
  and rtxo(1).tkval = ctxo.tkval                        // preserve value
else false
```